Юрий Магда



Ассемблер

Разработка и оптимизация Windows-приложений



+ CD-ROM

- Разработка высокопроизводительных программ
- Оптимизация программного кода с помощью ассемблера
- Программирование на ассемблере в Windows
- Применение ассемблера, встроенного в C++ и Delphi



Ю. С. Магда

Ассемблер

Разработка и оптимизация Windows-приложений

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1
М12

Магда Ю. С.

М12 Ассемблер. Разработка и оптимизация Windows-приложений. — БХВ-Петербург, 2003. — 544 с.: ил.

ISBN 5-94157-324-3

В книге рассматривается один из эффективных методов оптимизации программ — использование языка ассемблера, описана методика разработки отдельных модулей на нем для применения в программах на языках высокого уровня, показано, как с помощью ассемблера можно разработать полнофункциональные Windows-приложения. Особое внимание уделено оптимизации программ, написанных на языках высокого уровня, с помощью встроенного ассемблера. Для демонстрации методов и подходов выбраны наиболее популярные средства разработки — Microsoft Visual C++ .NET и Borland Delphi 7. В книгу включены примеры программного кода приложений, иллюстрирующие различные аспекты применения ассемблера. Исходные тексты программ содержатся на прилагаемом к книге компакт-диске.

Для профессиональных разработчиков программного обеспечения в Windows

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Анатолий Адаменко*
Зав. редакцией	*Григорий Добин*
Редактор	*Нина Седых*
Компьютерная верстка	*Татьяны Олоновой*
Корректор	*Вера Александрова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.07.03.
Формат 70×100[1]/16. Печать офсетная. Усл. печ. л. 43,86
Тираж 3000 экз. Заказ № 1064
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-324-3

Содержание

Введение

Эволюция средств проектирования программ в течение последних десятилетий способна удивить любого человека, занимающегося разработкой программного обеспечения. Особенно это касается написания программ для операционных систем семейства Windows. Современные инструментальные средства настолько развиты, что разработчик программного обеспечения может получить готовую программу с помощью нескольких щелчков мышью. Огромное количество книг, статей и исходных текстов программного кода посвящено проектированию программ на C, Pascal, Basic и других языках программирования. Язык низкого уровня — язык ассемблера — после "смерти" MS-DOS, казалось, доживает последние дни. Но вопреки прогнозам он не сошел с арены и продолжает весьма широко использоваться при разработке программ. В чем же секрет живучести этого языка?

Ответ довольно прост и может быть сформулирован одним предложением: язык ассемблера — это язык, на котором "говорит" процессор, и исчезнуть он может только вместе с исчезновением процессоров! По этой же причине ассемблер имеет одно фундаментальное преимущество перед языками высокого уровня: он самый быстрый. Большинство приложений, работающих в режиме реального времени, либо написаны на ассемблере, либо используют в критических участках кода ассемблерные модули.

Нередко можно слышать утверждения о том, что процесс разработки программ на языке ассемблера слишком трудоемок и отнимает массу времени. Еще одним препятствием для работы с ассемблером считают его сложность. И, наконец, в качестве аргумента приводится утверждение, что разработка приложений на ассемблере сильно затруднена из-за отсутствия современных инструментальных средств для проектирования и отладки.

Такие утверждения, вообще-то, не соответствуют действительности. Язык ассемблера не сложнее других языков программирования и довольно легко осваивается как опытными, так и начинающими программистами. Кроме того, в последние годы появились очень мощные инструментальные средства разработки программ на ассемблере, и это вынуждает по-другому взглянуть на процесс разработки программ на этом языке. Среди таких инструментальных средств проектирования можно назвать макроассемблер

MASM32, AsmStudio и NASM. Эти и другие аналогичные инструменты проектирования сочетают в себе гибкость и быстроту ассемблера и современный графический интерфейс. Многочисленные библиотеки функций, разработанные для ассемблера, приблизили этот язык по своим функциональным возможностям к средствам проектирования приложений на языках высокого уровня. Поэтому в настоящее время противопоставление ассемблера языкам высокого уровня не имеет под собой реальных оснований.

В книге рассматриваются вопросы применения ассемблера для разработки Windows-приложений. Материал книги раскрывает два относительно независимых аспекта применения ассемблера: как самостоятельного инструмента разработки небольших высокопроизводительных приложений и как встроенного средства в составе языков высокого уровня. Второй аспект применения ассемблера, вероятно, будет интересен программистам, работающим с языками высокого уровня, такими как Microsoft Visual C++ .NET и Delphi 7. Если вы обратили внимание, ведущие фирмы-производители, такие как Microsoft и Borland, постоянно совершенствуют встроенный ассемблер.

Автор должен сразу заметить, что книга не является учебником по ассемблеру или одному из языков высокого уровня и предполагает наличие у читателя определенных знаний в этих областях программирования.

Для успешной разработки программ в Windows желательно знать принципы работы приложений в этой операционной среде. От читателя не требуется глубоких знаний архитектуры Windows, поскольку все необходимые сведения приводятся в процессе анализа программного кода примеров.

Эта книга задумана как практическое пособие для программистов-разработчиков, желающих больше узнать о программировании на ассемблере. Программисты Visual C++ и Delphi смогут использовать ассемблер для разработки более совершенных процедур и модулей (а резервы здесь еще огромные!). Программисты, пишущие на ассемблере, возможно, также почерпнут полезные сведения для дальнейшей работы.

Материал книги включает много примеров с анализом программного кода. Автор считает, что любой теоретический вопрос должен подкрепляться примером программного кода. Это наиболее эффективный и быстрый способ научиться писать программы. Все примеры программ являются работоспособными и проверены автором. Автор сознательно избегает длинных и сложных программ, поскольку при их анализе легко теряются ключевые моменты, ради которых эти программы и были разработаны. Каждый пример построен таким образом, чтобы его можно было легко адаптировать или модифицировать для дальнейшего использования.

В качестве инструментов разработки выбраны продукты фирм Microsoft и Borland — Visual Studio .NET и Delphi 7. Эти средства проектирования являются наиболее популярными в среде российских программистов. Приме-

ры программного кода написаны для использования именно с этими языками программирования. Автор не видит смысла использовать компиляторы более ранних версий.

Что касается примеров на ассемблере, то здесь в основном используется макроассемблер MASM, хотя в отдельных случаях автор приводит код и для Турбо ассемблера фирмы Borland TASM 5.0. Программный код, написанный с использованием MASM, легко адаптируется для работы с Турбо ассемблером TASM.

Читателям автор рекомендует использовать ассемблер MASM32, который включает в себя компилятор ML версии 6.14 и компоновщик LINK версии 5.12 фирмы Microsoft.

Во всех примерах используется упрощенный синтаксис языка ассемблера и минимум высокоуровневых конструкций языка. Автор не приводит детального описания компилятора MASM, а упоминает лишь те сведения, которые необходимы для работы. Читатели, желающие углубить свои знания о работе компилятора, без труда смогут найти массу информации по этому вопросу в других источниках.

Автор стремится рассматривать материал в определенной логической последовательности, избегая как нагромождения программного кода, так и излишнего теоретизирования. Автор отдает себе отчет в том, что в одной книге сложно рассмотреть все аспекты программирования в Windows, тем не менее он надеется, что материал книги окажется достаточно полезным для программистов.

Структура книги

Книга задумана как практическое пособие по оптимизации программного кода на ассемблере. Язык ассемблера может применяться как самостоятельный инструмент разработки приложений, так и в качестве встроенного средства разработки языков высокого уровня. Оба эти аспекта подробно рассмотрены в книге. Структура книги построена таким образом, чтобы можно было изучить материал как выборочно по отдельным главам, так и последовательно, начиная с первой главы. Это удобно, т. к. позволяет различным категориям читателей выбирать тот материал, который им более всего интересен. Как начинающие, так и опытные программисты смогут найти в ней необходимые сведения.

Автор делает упор на практический аспект применения полученной информации. Многочисленные примеры позволяют лучше понять принципы разработки и оптимизации программ, а необходимый теоретический материал дается в контексте приводимых примеров. Описание программных средств ассемблера и языков высокого уровня дается в том объеме, в каком это не-

обходимо для понимания материала. Автор считает излишним помещать в книге полный справочный материал по компиляторам и компоновщикам, дублируя многочисленные литературные источники и фирменные руководства.

Примеры программ построены таким образом, чтобы продемонстрировать ключевые аспекты техники применения ассемблера. В программах на ассемблере не используются ни макросы, ни высокоуровневые конструкции этого языка, за исключением одного оператора `invoke`.

Примеры программ на языках высокого уровня со встроенным ассемблером используют наиболее часто применяемые компоненты и элементы управления. Как правило, в таких примерах основной упор делается на интерфейс ассемблерной процедуры и основной программы.

Большинство программ, написанных на ассемблере, представляют собой полнофункциональные графические приложения Windows, поэтому при анализе таких примеров объясняются ключевые аспекты функционирования Windows-приложений. Программный код примеров подобран так, чтобы его можно было применить в собственных разработках. Автор надеется, что предоставленные примеры задач будут действительно полезны программистам.

Книга состоит из 6 глав, краткие сведения о каждой из них приведены далее.

- *Глава 1* "Разработка высокоэффективного программного кода". В этой главе рассматриваются общие вопросы оптимизации программ. Как известно, существует множество различных способов сделать работу программы более эффективной. В этом случае важно выбрать наиболее подходящий вариант. Такой сравнительный анализ и проводится в этой главе.
- *Глава 2* "Основы программирования на языке ассемблера". Материал этой главы посвящен наиболее важным аспектам языка ассемблера с точки зрения повышения производительности программ. Здесь рассматриваются алгоритмы обработки математических выражений, массивов данных и строк. Наряду с этим большое внимание уделено работе с подпрограммами, их структуре. Для демонстрации примеров функционирования программного кода этой главы используются 32-разрядные консольные приложения.
- *Глава 3* "Интерфейс с языками высокого уровня". В этой главе основное внимание уделено оптимизации наиболее важных конструкций языков высокого уровня — циклов и условных операторов. Проводится анализ математических выражений, написанных на языках высокого уровня, и возможная замена таких выражений на их ассемблерные аналоги. В рассматриваемом контексте этот материал дополняет *главу 2*. Здесь же рассматриваются наиболее общие вопросы интерфейса отдельных процедур, полностью написанных на ассемблере, с языками высокого уровня.

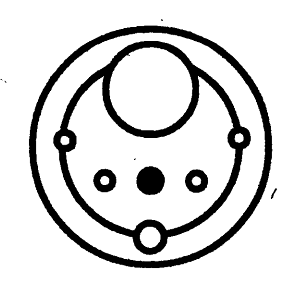
Как и в предыдущей главе, для демонстрации материала приводятся многочисленные примеры.

- *Глава 4* "Программирование приложений в Windows на языке ассемблера: первые шаги". В этой главе рассматривается архитектура 32-разрядных приложений в операционных системах Windows, а также общие вопросы разработки таких приложений на языке ассемблера. На примере каркасного приложения анализируются различные аспекты создания графических приложений. В конце главы приводится исходный текст классического приложения Windows, написанного на ассемблере.
- *Глава 5* "Программирование на ассемблере в Windows: от простого к сложному". В этой главе раскрываются наиболее существенные аспекты разработки приложений на ассемблере. Здесь рассмотрены вопросы программирования ввода-вывода текстовых данных, периферийных устройств (клавиатура и мышь), применения элементов управления, в том числе и диалоговых окон. Теоретический материал подкреплен многочисленными примерами. Особое внимание уделено разработке и использованию библиотек динамической компоновки.
- *Глава 6* "Встроенный ассемблер языков высокого уровня: принципы использования". Глава посвящена применению встроенного ассемблера языков высокого уровня для достижения высокой производительности работы приложений. Удобство и легкость использования встроенного ассемблера трудно переоценить. По своим возможностям он практически эквивалентен автономным средствам разработки, таким как макроассемблер MASM или Турбо ассемблер TASM 5.0, но в отличие от них не требует ни отдельной компиляции, ни компоновки блоков ассемблерного кода. Это очень важно для быстрой разработки и отладки приложений. Как и в остальных главах, теория подкреплена практическими примерами программ.

Материал книги дополнен справочником по системе команд процессоров Intel. Поскольку полная система команд насчитывает несколько сотен наименований, то приведены только наиболее часто используемые команды. Значительную помощь читателю окажет и прилагаемый CD, на котором записаны все примеры программ, приведенных в книге.

Автор благодарит всех, кто принял участие в создании этой книги. Особенно признателен жене Юлии за поддержку и помощь в процессе работы над книгой. Огромная благодарность также сотрудникам издательства "БХВ-Петербург" за подготовку материалов книги к изданию.

Глава 1



Разработка высокоэффективного программного кода

Эта глава посвящена общим вопросам *оптимизации* программного обеспечения. Термин "оптимизация" применительно к процессу разработки и отладки программ подразумевает улучшение каких-либо характеристик работы программного продукта. Под этим термином подразумевают часто и комплекс мер по улучшению показателей производительности программы.

Сам процесс оптимизации программного обеспечения может выполняться как программистом (*ручная оптимизация*), так и в автоматическом режиме *компилятором* той среды разработки, в которой производится отладка приложения. Возможен и вариант, когда программист использует программу-отладчика третьей фирмы для выполнения отладки и оптимизации.

Большинство разработчиков понимает, что в условиях жесткой конкуренции вопрос производительности является важнейшим условием успеха или неудачи программы на рынке программных продуктов. Без серьезной работы над улучшением производительности *программного кода* нельзя обеспечить конкурентоспособность приложения. Хотя все осознают необходимость и важность процесса оптимизации программного обеспечения, эта тема до сих пор является противоречивой и дискуссионной. Все споры вокруг этого процесса в основном затрагивают один вопрос: так ли уж необходимо программисту заниматься ручной оптимизацией своего приложения, если для этого есть готовые аппаратно-программные средства?

Часть программистов считает, что улучшить производительность работы приложения без использования средств отладки самого компилятора нельзя, тем более что все современные компиляторы имеют *встроенные* (built-in) средства оптимизации программного кода. Отчасти это правда. На сегодняшний день все без исключения программные средства разработки предусматривают использование оптимизационных алгоритмов при генерации исполняемого модуля.

Можно полностью положиться на компилятор ("все сделано до нас"), который сгенерирует для вас оптимальный код, и вообще не заниматься улучшением качества программы. При этом в целом ряде случаев может и не понадобиться никаких доработок и улучшений. Например, при создании небольших офисных приложений или утилит тестирования сети оптимизация обычно не нужна.

Однако в большинстве случаев обойтись без ручной оптимизации и полагаться только на стандартные возможности компиляторов нельзя. С проблемой улучшения производительности, хотите вы этого или нет, вам неизбежно придется столкнуться при разработке более или менее серьезных приложений, например *баз данных*, любых клиент-серверных или сетевых приложений, причем оптимизирующий компилятор той среды, в которой вы работаете, в большинстве случаев значительного выигрыша вам не обеспечит.

Если программист разрабатывает программы, работающие в *реальном времени*, такие как *драйверы устройств*, *системные службы* или промышленные приложения, то без очень серьезной работы по ручной доводке кода до оптимальной производительности задача написания программы просто не будет выполнена. И дело здесь не в том, что средства разработки несовершенны и не обеспечивают того уровня оптимизации, какой от них требуется. Любая более или менее серьезная программа имеет столько взаимосвязанных параметров, что ни одно средство разработки не улучшит ее так, как это может сделать сам программист. Процесс оптимизации программ сродни скорее искусству, чем "чистому" программированию, и трудно поддается *алгоритмизации*.

Улучшение производительности программ — процесс обычно трудоемкий, занимающий значительную часть времени. Хочется отметить, что не существует единого критерия оптимизации. Более того, сам процесс оптимизации довольно противоречив. Например, если добиться уменьшения *объема памяти*, используемого программой, то за это придется расплатиться потерей быстродействия работы программы.

Ни одна программа не может быть одновременно сверхбыстрой, сверхмалой по размеру и полнофункциональной для пользователя. К этому можно сколь угодно приближаться, но идеального приложения вам получить никогда не удастся.

Хорошие программы обычно сочетают те или иные качества в разумных пропорциях, в зависимости от того, что важнее: скорость выполнения, размер программы (как *файла* приложения, так и объема памяти, занимаемого работающим приложением) или удобство *интерфейса пользователя*.

Для многих офисных приложений очень важным показателем является удобство интерфейса пользователя и как можно более высокая функциональность. Например, для пользователя программы электронного телефон-

ного справочника тот факт, что программа работает на 10% быстрее или медленнее, особого значения не имеет. Размер такой программы, в принципе, не критичен и также не имеет особого значения, т. к. объем современных жестких дисков достаточно большой, чтобы поместить десятки и даже сотни таких электронных справочников. Программе может быть необходимо десятки мегабайт оперативной памяти для работы — это тоже сейчас не проблема. Но вот возможность удобной манипуляции данными для пользователя будет очень важной.

Если приложение использует клиент-серверную модель обработки данных и взаимодействия с пользователем, как, например, большинство сетевых приложений, то критерии оптимизации здесь будут несколько иными. На первое место могут выйти проблемы использования памяти (особенно для серверной части приложения) и оптимизации сетевого взаимодействия с клиентской частью.

Приложения, работающие в режиме реального времени, критичны по синхронизации получения, обработки и, возможно, передачи данных за приемлемые интервалы времени. Подобные программы требуют, как правило, оптимизации по загрузке процессора и синхронизации с системными службами операционной системы. Если вы — системный программист и разрабатываете драйверы или сервисы для работы с операционной системой, например с Windows 2000, то неэффективный программный код в лучшем случае только замедлит работу всей операционной системы, а о худших последствиях можно только догадываться.

Как видим, повышение производительности программ зависит от многих факторов и в каждом конкретном случае определяется тем, что эта программа должна делать.

Рассмотрим теперь более подробно, как можно выполнить оптимизацию программ, и проведем небольшой сравнительный анализ различных методов повышения производительности выполнения приложений.

Простейший способ заставить приложения работать быстрее — это повысить вычислительную мощь компьютера за счет установки более производительного процессора или увеличения объема памяти, т. е. сделать *апгрейд* (upgrade) аппаратной части. В этом случае проблема производительности будет решена сама собой.

Если вы сторонник такого подхода, то скорей всего окажетесь в тупике, т. к. будете все время зависеть от аппаратных решений. К слову сказать, многие ожидания насчет производительности новых поколений процессоров, новых типов памяти и архитектур *системных шин* оказываются явно преувеличенными. Их производительность на практике оказывается ниже задекларированной фирмами-изготовителями. Так, например, новые микросхемы памяти, как правило, превосходят своих предшественников по объему хранимых данных, но отнюдь не по быстродействию. Производительность жестких дисков также растет медленнее, чем их объем.

Если вы разрабатываете коммерческое приложение, то должны учитывать, что у пользователя может не быть самых последних моделей процессора и быстродействующей памяти. К тому же, далеко не все пользователи горят желанием выложить деньги на новый компьютер, если их вполне устраивает то, что у них уже есть.

Поэтому вряд ли стоит полагаться всерьез на решение проблем с программным обеспечением при помощи одной только закупки нового оборудования.

Далее мы будем рассматривать только алгоритмические и программные методы повышения производительности работы приложений.

Оптимизация может проводиться по следующим направлениям:

- тщательная проработка алгоритма разрабатываемой программы;
- учет существующих аппаратных средств компьютера и использование их оптимальным образом;
- использование средств *языка высокого уровня* той среды, в которой разрабатывается приложение;
- использование языка низкого уровня *ассемблера*;
- учет специфических особенностей процессора.

Рассмотрим более подробно каждое из этих направлений.

1.1. Оптимизация алгоритма разрабатываемой программы

Этап разработки алгоритма вашего приложения — самый сложный во всей цепочке жизненного цикла программы. От того, насколько глубоко продуманы все аспекты вашей задачи, во многом зависит успех ее реализации в виде программного кода. В общем случае изменения в структуре самой программы дают намного больший эффект, чем тонкая настройка программного кода. Идеальных решений не бывает, и разработка алгоритма приложения всегда сопровождается ошибками и недоработками. Здесь важно найти узкие места в алгоритме, наиболее влияющие на производительность работы приложения.

Кроме того, как показывает практика, почти всегда можно найти способ улучшить уже разработанный алгоритм программы. Конечно, лучше всего тщательно разработать алгоритм в начале проектирования, чтобы избежать в дальнейшем многих неприятных последствий, связанных с доработкой фрагментов программного кода в течение короткого промежутка времени. Не жалейте времени на разработку алгоритма приложения — это избавит вас от головной боли при отладке и тестировании программы и сэкономит время.

Следует иметь в виду, что алгоритм, эффективный с точки зрения производительности программы, никогда не соответствует требованиям постановки задачи на все 100% и наоборот. Неплохие с точки зрения структуры и читабельности алгоритмы, как правило, не эффективны в плане реализации программного кода. Одна из причин — стремление разработчика упростить общую структуру программы за счет использования везде, где только можно, высокоуровневых вложенных структур для вычислений. Упрощение алгоритма в этом случае неизбежно ведет к снижению производительности программы.

В начале разработки алгоритма довольно сложно оценить, каким будет программный код приложения. Чтобы правильно разработать алгоритм программы, необходимо следовать нескольким простым правилам:

1. Тщательно изучить задачу, для которой будет разработана программа.
2. Определить основные требования к программе и представить их в формализованном виде.
3. Определить форму представления входных и выходных данных и их структуру, а также возможные ограничения.
4. На основе этих данных определить программный вариант (или модель) реализации задачи.
5. Выбрать метод реализации задачи.
6. Разработать алгоритм реализации программного кода. Не следует путать алгоритм решения задачи с алгоритмом реализации программного кода. В общем случае, они никогда не совпадают. Это самый ответственный этап разработки программного обеспечения!
7. Разработать исходный текст программы в соответствии с алгоритмом реализации программного кода.
8. Провести отладку и тестирование программного кода разработанного приложения.

Не следует воспринимать эти правила буквально. В каждом конкретном случае программист сам выбирает методику разработки программ. Некоторые этапы разработки приложения могут дополнительно детализироваться, а некоторые вообще отсутствовать. Для небольших задач достаточно разработать алгоритм, слегка подправить его для реализации программного кода и затем отладить.

При создании больших приложений, возможно, понадобится разрабатывать и тестировать отдельные фрагменты программного кода, что может потребовать дополнительной детализации программного алгоритма.

Для правильной алгоритмизации задач программисту могут помочь многочисленные литературные источники. Принципы построения эффективных алгоритмов достаточно хорошо разработаны. Имеется немало хорошей литературы по этой теме, например книга Д. Кнута "Искусство программирования".

1.2. Оптимизация с учетом аппаратных средств компьютера

Обычно разработчик программного обеспечения стремится к тому, чтобы производительность работы приложения как можно меньше зависела от аппаратуры компьютера. При этом следует принимать во внимание наихудший вариант, когда у пользователя вашей программы будет далеко не самая последняя модель компьютера. В этом случае "ревизия" работы аппаратной части часто позволяет найти резервы для улучшения работы приложения.

Первое, что нужно сделать, — проанализировать производительность компьютерной периферии, на которой должна работать программа. В любом случае знание того, что работает быстрее, а что медленнее, поможет при разработке программы. Анализ пропускной способности системы позволяет определить узкие места и принять правильное решение.

Различные устройства компьютера имеют разную пропускную способность. Наиболее быстрыми из них являются процессор и оперативная память, относительно медленными — жесткий диск и CD-привод. Самыми медленными являются принтеры, плоттеры и сканеры.

Основная часть Windows-приложений разрабатывается с графическим пользовательским интерфейсом и активно использует графические возможности компьютера. В этом случае при разработке приложения необходимо учесть пропускную способность системной шины и графической подсистемы компьютера.

Практически все приложения активно используют ресурсы жесткого диска. В большинстве случаев производительность дисковой подсистемы оказывает значительное влияние на работу приложения. Если программа интенсивно использует ресурсы жесткого диска, например, часто выполняет запись-перемещение файлов, то при относительно медленном жестком диске неизбежно возникнут проблемы с производительностью.

Приведем другой пример. Преимущественное использование *регистров* центрального процессора может повысить производительность программы за счет уменьшения обмена по системной шине, как это случается при работе с оперативной памятью. Во многих случаях повысить производительность приложения можно путем *кэширования* данных. Это может помочь при дисковых операциях, работе с мышью, устройством печати и т. д.

Если вы разрабатываете коммерческое приложение, то обязательно выясните, с какой наихудшей аппаратной конфигурацией будет работать ваша программа. Все мероприятия по оптимизации проводите с учетом именно такой конфигурации аппаратных средств.

1.3. Оптимизация с использованием средств языка высокого уровня

Использование такого метода оптимизации обычно связано с анализом программного кода на предмет выявления узких мест (bottlenecks) в процессе функционирования приложения. Обычно точки, в которых программа значительно замедляет работу, выявить не так просто. В этом разработчику могут помочь специальные программы, называемые *профайлерами* (profiler). Их назначение — определить производительность приложений, помочь при отладке и выявить точки программы, в которых производительность падает. Одной из наилучших программ этого класса является Intel VTune Performance Analyzer. Можно рекомендовать использовать именно эту программу для отладки и оптимизации приложений.

Встроенные средства отладки имеются и в языках высокого уровня. Современные компиляторы позволяют обнаруживать ошибки, однако они не предоставляют никакой информации об эффективности выполнения того или иного участка программы. Вот почему желательно иметь под рукой какой-нибудь хороший профайлер.

Многие программисты предпочитают вести отладку приложений вручную. Это не самый худший вариант, если вы хорошо представляете себе работу приложения. В любом случае, как бы вы не проводили отладку, полезно обратить внимание на некоторые моменты, влияющие на производительность работы приложения:

- количество вычислений, выполняемых программой. Одним из условий повышения производительности приложения является уменьшение объема вычислений. Работающая программа не должна вычислять одно и то же значение дважды. Вместо этого она должна рассчитать каждое значение один раз и сохранить его в памяти для повторного использования. Существенного повышения быстродействия приложения можно добиться, если преобразовать математические вычисления в обращения к таблицам, которые могут быть сгенерированы заранее;
- использование математических операций. Любое приложение, так или иначе, использует математические операции. Анализ эффективности вычислений довольно сложен и в каждом конкретном случае зависит от многих факторов. Выигрыш в производительности может дать использование более простых арифметических операций для вычислений. Везде, где только можно, операции умножения и деления следует заменить соответствующим блоком команд сложения/вычитания. Если в программе используются *операции с плавающей точкой*, то старайтесь не использовать команды обработки целых чисел, т. к. они замедляют работу приложения. Еще один нюанс: используйте по возможности как можно меньше операций деления. Производительность заметно падает и при

использовании математических операций в *циклах*. Операции умножения на степень двойки можно заменить командами сдвига влево;

- использование циклических вычислений и вложенных структур. Речь идет об использовании циклов WHILE, FOR, SWITCH, IF. Циклические вычисления упрощают структуру программы, но уменьшают производительность. Внимательно просматривайте программный код на предмет поиска вложенных вычислений с использованием циклических структур. Полезно помнить несколько правил, которые помогают при оптимизации циклов:

  - никогда не следует делать в цикле то, что можно выполнить за его пределами;
  - по возможности избавляйтесь от команд передачи управления внутри циклов.

  Вынос за пределы цикла даже одного или двух операторов способен улучшить показатели производительности. Эффективной работе приложения способствуют и такие действия, как вычисление неизменяющихся величин за пределами циклов; разворачивание циклов и объединение отдельных циклов, выполняемых одно и то же количество раз, в единый цикл. Следует избегать использования большого количества команд в теле цикла. Старайтесь применять меньше вызовов подпрограмм из тела цикла, т. к. вычисление эффективных *адресов процедур* может значительно замедлить работу процессора.

  Полезным в плане улучшения производительности будет и уменьшение количества передач управления в программе. Для этого можно, например, преобразовать условные переходы таким образом, чтобы условие перехода становилось истинным значительно реже, чем условие его отсутствия. Полезно также перемещать условия общего характера в начало ветвления последовательности переходов. Если в программе есть вызовы, после которых следует возврат в программу, то желательно преобразовать такие вызовы в переходы.

  Подытоживая этот пункт, можно сделать следующий вывод: желательно избавляться от переходов и вызовов везде, где только можно, особенно в тех точках программы, где на быстродействие влияет только процессор. Для этого программа должна быть организована так, чтобы она исполнялась прямым (линейным) последовательным образом с минимальным числом точек переходов;

- реализация механизма *многопоточности* (multithreading). Правильное использование этого механизма в программе может повысить ее производительность, а неправильное — наоборот, замедлить. Как показывает практика, использование многопоточности эффективно применять для больших приложений, небольшие же программы начинают работать медленнее. Возможность разделения выполняемого процесса на несколько потоков заложена в архитектуре операционных систем Windows.

Многопоточность можно использовать для оптимизации программ. Необходимо помнить, что каждый поток требует дополнительных ресурсов памяти и процессора, поэтому при слабой аппаратной поддержке (медленный процессор или недостаточный объем памяти) все усилия по улучшению производительности этим методом могут оказаться неэффективными;

- выделение часто повторяющихся однотипных вычислений в отдельные подпрограммы (процедуры). Очень распространенным является мнение, что использование подпрограмм всегда повышает производительность приложений, т. к. позволяет многократно применить один и тот же фрагмент кода для выполнения однотипных вычислений в разных местах программы. С точки зрения читабельности программы и понимания алгоритма работы, это действительно так. Но "с точки зрения процессора" выполнение программы по линейному алгоритму всегда (!) эффективнее, чем использование процедур. Каждый раз, когда вы используете процедуру, выполняется переход по другому адресу памяти с сохранением адреса возврата в основную программу в стеке. Это всегда вызывает замедление выполнения программы. Все сказанное не означает, что нужно отказаться от использования подпрограмм, нужно лишь разумно применять их в своих разработках.

1.4. Оптимизация с использованием языка низкого уровня ассемблера

Использование языка ассемблера — это один из наиболее действенных методов оптимизации программ, и во многом методы, используемые для повышения производительности, схожи с теми, что используются в языках высокого уровня. Однако язык ассемблера предоставляет программисту и ряд дополнительных возможностей. Я не буду повторять то, что уже сказано в контексте оптимизации с использованием средств языков высокого уровня, а выделю методы, свойственные только ассемблеру.

- Использование языка ассемблера во многом решает проблему избыточности программного кода. Ассемблерный код более компактен, чем его аналог на языке высокого уровня. Чтобы убедиться в этом, достаточно сравнить дизассемблированные листинги одной и той же программы, написанной на ассемблере и на языке высокого уровня. Сгенерированный компилятором языка высокого уровня ассемблерный код даже с использованием опций оптимизации не устраняет избыточность кода приложения. В то же время язык ассемблера позволяет разрабатывать короткий и эффективный код.
- Программный модуль на ассемблере обладает, как правило, более высоким быстродействием, чем написанный на языке высокого уровня. Это

связано с меньшим числом команд, требуемых для реализации фрагмента кода. Меньшее число команд быстрее выполняется центральным процессором, что, соответственно, повышает производительность программы.

- Можно разрабатывать отдельные модули полностью на ассемблере и присоединять их к программам на языке высокого уровня. Также можно использовать мощные встроенные средства языков высокого уровня для написания ассемблерных процедур непосредственно в теле основной программы. Такие возможности предусмотрены во всех языках высокого уровня. Эффективность использования встроенного ассемблера может быть очень высока. Встроенный ассемблер позволяет добиваться максимального эффекта при оптимизации математических выражений, программных циклов и обработки массивов данных в основной программе.

1.5. Оптимизация с учетом специфических особенностей процессора

В основе оптимизации с учетом специфических особенностей процессора лежат особенности архитектуры конкретного типа процессора Intel. Он представляет собой расширение варианта оптимизации с использованием языка ассемблера.

Мы будем рассматривать варианты оптимизации только для процессоров Pentium. Каждая последующая модель процессора обычно имеет дополнительные архитектурные улучшения по сравнению с предыдущей. В то же время все модели процессоров Pentium имеют и общие характеристики. Поэтому оптимизация на уровне процессора может проводиться как на основе общих характеристик всего семейства, так и с учетом особенностей каждой модели.

Оптимизация программного кода на уровне процессора позволяет повысить производительность не только приложений на языке высокого уровня, но и процедур, написанных на ассемблере. Программисты, пишущие на языках высокого уровня, практически незнакомы с этой методикой оптимизации и используют ее относительно редко, хотя возможности ее безграничны. Разработчики ассемблерных программ и процедур иногда используют возможности новых типов процессоров.

Должен заметить, что и более ранние типы процессоров Intel включают дополнительные команды, которые редко применяются разработчиками, но позволяют сделать программный код более эффективным.

Какие возможности процессора можно использовать для оптимизации? Прежде всего, будет полезным выравнивание данных и адресов по границам

32-разрядных слов. Кроме того, все процессоры, начиная с 80386, обладают расширенными вычислительными возможностями, которые можно использовать для оптимизации программ. Такие возможности появились благодаря дополнительным командам и расширению возможностей адресации операндов. Производительность программ можно увеличить, используя:

- команды пересылки с нулевым или знаковым расширением (`movzx` или `movsx`);
- установки в байте значений "истина" или "ложь" в зависимости от содержимого *флагов* центрального процессора, что позволяет избавиться от команд условного перехода (`setz`, `setc` и т. д.);
- команды проверки, установки, сброса и сканирования битов (`bt`, `btc`, `btr`, `bts`, `bsp`, `bsr`);
- обобщенную индексную адресацию и режимы адресации с масштабированием индексов;
- быстрое умножение при помощи команды `lea` с использованием масштабированной индексной адресации;
- перемножение 32-разрядных чисел и деление 64-разрядного числа на 32-разрядное;
- операции для обработки многобайтных массивов данных и строк.

Команды процессора, выполняющие копирование и перемещение массивов многобайтных данных, требуют меньше циклов процессора, чем классические команды этой группы. Начиная с процессоров ММХ, появились комплексные команды, сочетающие в себе несколько функций, выполняемых отдельными командами. Значительно расширилась группа команд для выполнения битовых операций. Эти команды также являются комплексными и позволяют выполнить несколько операций одновременно. Мы рассмотрим возможности, предоставляемые такими командами, в *главе 6*, когда будем анализировать встроенные средства языков высокого уровня.

Как вы уже убедились, большие возможности для оптимизации программ кроются в правильном использовании особенностей аппаратной архитектуры процессора. Это довольно сложная сфера, т. к. требует знания методов обработки данных и выполнения команд процессора на уровне аппаратной части. Могу с уверенностью утверждать, что возможности для оптимизации здесь безграничны.

Естественно оптимизация на уровне процессора имеет свои особенности. Например, если программа должна работать с процессорами нескольких поколений, то оптимизация должна учитывать общие особенности всех этих устройств.

Здесь представлен далеко не полный перечень возможных вариантов оптимизации программного кода приложений. Как очевидно, большие резервы

для повышения эффективности работы программы кроются в самой программе. В книге основное внимание будет уделено оптимизации программного кода с использованием языка ассемблера, поэтому далее рассмотрим более детально, как решаются такие задачи.

1.6. Ассемблер и оптимизация программ в деталях

Язык ассемблера, как средство улучшения производительности приложений, написанных на языках высокого уровня, используется очень широко. Разумное сочетание в одном приложении модулей, написанных на языке высокого уровня и на ассемблере, позволяет достичь как высокого быстродействия работы программы, так и уменьшения размера исполняемого кода. В настоящее время такое сочетание используется настолько часто, что фирмы-разработчики компиляторов уделяют особое внимание интерфейсу программ на языках высокого уровня с ассемблером. Современные компиляторы языков высокого уровня имеют, как правило, встроенный ассемблер.

На практике применяются два варианта совместного использования ассемблера и языков высокого уровня. В первом случае используется отдельный файл объектного модуля, в котором располагается одна или несколько процедур обработки данных. Вызов процедур осуществляется программой, написанной с использованием высокоуровневой среды разработки, например Delphi или Visual C++.

В исходном тексте приложения на языке высокого уровня ассемблерная процедура объявляется соответствующим образом, после чего ее можно вызвать из любой точки основной программы. Внешний объектный модуль на ассемблере присоединяется к основной программе на этапе компоновки.

Файл с исходным текстом процедуры обычно имеет расширение ASM и компилируется одним из распространенных пакетов, таких как Microsoft Macro Assembler (MASM), Borland Turbo Assembler (TASM 5.0) или Netwide Assembler (NASM). Последний компилятор превосходит первые два по своим возможностям, однако так сложилось, что в странах бывшего СНГ наиболее популярными являются все же компиляторы MASM и TASM.

Преимущества отдельно компилируемых модулей на ассемблере — это возможность использования программного кода в приложениях, написанных на разных языках и даже в разных операционных средах, и независимость процесса разработки и отладки программного кода процедур. К недостаткам, пожалуй, можно отнести некоторые сложности компоновки разработанного модуля с основной программой на языке высокого уровня. При этом необходимо четко представлять механизм вызова внешних процедур и передачи параметров в вызываемую процедуру. Преимущества такого подхода — многократное использование разработанных на ассемблере объект-

ных модулей или библиотек функций. В этом случае программист должен позаботиться об интерфейсе ассемблерного модуля с программой, написанной на языке высокого уровня. Вопросы компоновки ассемблерных модулей и программ на языках высокого уровня подробно будут рассматриваться в *главе 3*.

Второй вариант совместного использования ассемблера и языков высокого уровня основан на применении встроенного ассемблера. Разработка процедур на встроенном ассемблере удобна, в первую очередь, благодаря быстроте отладки. Так как процедура разрабатывается в теле основной программы, то не требуется специальных средств для компоновки такой процедуры с вызывающей программой. Не нужно также заботиться о порядке передачи параметров в вызываемую процедуру и о восстановлении стека. К недостаткам этого метода оптимизации можно отнести определенные ограничения, которые накладывает среда программирования на работу ассемблерных модулей, а также то, что процедуры, разработанные на встроенном ассемблере, нельзя преобразовать во внешние отдельно используемые модули.

Все современные средства разработки ассемблерных программ имеют в своем составе интегрированный отладчик, так же как и языки высокого уровня. И хотя такой отладчик предоставляет меньший уровень сервиса по сравнению с языками высокого уровня, он вполне достаточен для анализа программного кода.

1.7. Использование ассемблера для разработки Windows-приложений

Несмотря на то, что ассемблер воспринимается многими программистами только как вспомогательное средство для улучшения программ, его значение как самостоятельного средства разработки высокоэффективных приложений в последнее время очень изменилось.

До сих пор существует некий стереотип, касающийся разработки приложений на ассемблере. Среди многих программистов, пишущих на языках высокого уровня, бытует мнение о сложности ассемблера, плохой структурируемости программного обеспечения и плохой переносимости кода при переходе на другие платформы. Возможно, многие помнят разработку программ на ассемблере в MS-DOS, что действительно требовало немалых усилий. Кроме того, отсутствие в то время современных средств программирования на ассемблере замедляло разработку сложных проектов.

В последнее время ситуация изменились благодаря появлению принципиально новых и эффективных средств быстрой разработки на языке ассемблера. Специально для этого были разработаны мощные *средства быстрого проектирования* (Rapid Application Development — RAD), такие как MASM32, Visual Assembler, RADASM. Размер и быстродействие оконного

приложения SDI (single-document interface), написанного на ассемблере, просто впечатляет!

Такие средства разработки имеют, как правило, компиляторы ресурсов, большие библиотеки готовых к использованию функций и мощные средства отладки. Можно смело утверждать, что разработка программ на ассемблере стала столь же легкой, как и на языках высокого уровня.

Основная причина, по которой ассемблер не применялся массово для разработки программ, — отсутствие средств быстрого проектирования — исчезла. Какие приложения можно проектировать на ассемблере? Проще ответить на другой вопрос — что не следует писать на ассемблере? Небольшие и средние по объему 32-разрядные приложения для Windows можно целиком написать на ассемблере. Однако если разрабатывать сложную программу, требующую применения самых современных технологий, то лучше использовать языки высокого уровня с последующей оптимизацией отдельных участков кода на ассемблере.

Существует еще одна проблема использования ассемблера, связанная с тем, что этот язык рассчитан на разработку *процедурно-ориентированных* приложений и не использует методы *объектно-ориентированного* программирования (ООП). Именно это приводит к некоторым ограничениям при использовании ассемблера. Тем не менее это никак не мешает применять язык ассемблера для написания классических Windows-приложений процедурно-ориентированного типа.

Современные средства разработки программ на ассемблере не только позволяют создать графический интерфейс пользователя, но и сохраняют фундаментальное преимущество ассемблера: фантастически малый размер исполняемого модуля. Короткие быстрые приложения на ассемблере находят применение там, где размеры кода и его быстродействие являются критическими параметрами. Сферами применения таких приложений являются системы реального времени, системные утилиты и программы, а также драйверы устройств.

Программы на ассемблере управляют как периферийным оборудованием персонального компьютера (ПК), так и нестандартными устройствами, присоединенными к ПК. Минимальные размеры программного кода обеспечивают высокое быстродействие работы таких устройств. Приложения реального времени используются повсеместно в системах управления в промышленности, научных и лабораторных исследованиях, в военных разработках.

Особенность системных программ и утилит состоит в том, что они очень тесно взаимодействуют с операционной системой, и скорость выполнения таких приложений может существенно повлиять на общую производительность всей системы. Это в значительной степени относится и к разработке драйверов периферийных устройств компьютера и системных служб.

Средства разработки на ассемблере позволяют создавать и быстрые утилиты командной строки (*консольные* приложения). Использование в таких утилитах системных вызовов Windows позволяет выполнить очень многие сложные функции (копирование файлов, функции поиска и сортировки, обработка и анализ математических выражений и т.д.) с очень высоким быстродействием.

Другой важной областью применения ассемблера является разработка драйверов нестандартных и специализированных устройств, управляемых при помощи ПК. В таких случаях использование программ на языке ассемблера будет очень эффективным. Примеров такого применения ассемблера можно привести много. Это и системы обработки данных на базе ПК с использованием выносных устройств, одноплатные компьютеры с флэш-памятью, системы диагностики и тестирования различного оборудования.

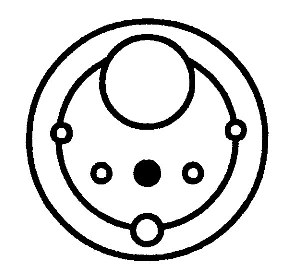
Подробно мы рассмотрим программирование Windows-приложений на ассемблере в *главах 4—5*.

Необходимо также упомянуть еще об одном аспекте применения языка ассемблера, достаточно экзотическом, но тем не менее используемом. Основная программа пишется на ассемблере, а вспомогательные модули — на любом другом языке, например на C++ или Pascal. При этом основная программа использует, как правило, мощные библиотечные функции языка высокого уровня, например математические или строковые. Кроме того, если для разработки интерфейса используются вызовы WIN API (Application Programming Interface), то программа получится очень мощной. Конечно, написание таких программ требует от программиста незаурядных знаний ассемблера и языков высокого уровня.

Мы рассмотрели далеко не все методы улучшения качества программного обеспечения. Существует масса трюков и ухищрений, которыми пользуются опытные программисты для улучшения показателей производительности.

Оптимизация программ, как уже упоминалось, процесс творческий, и каждый программист весьма индивидуален в выборе методики отладки своих программ.

Глава 2



Основы программирования на языке ассемблера

В этой главе рассматриваются те аспекты программирования на языке ассемблера, которые делают его действительно полезным и эффективным для написания приложений. Основное внимание будет уделено ключевым моментам, таким как программирование *математических функций*, обработка *массивов данных* и *строковые операции*. Использование ассемблера как самостоятельного инструмента разработки и как средства оптимизации невозможно без знания принципов построения подпрограмм (*процедур*), которые играют важную роль, особенно при компоновке с языками высокого уровня. Мы не будем изучать язык ассемблера, что называется, с нуля. Для этого существует много хороших учебников, в которых детально описана система команд и синтаксис ассемблера для процессоров фирмы Intel.

Эта глава раскрывает основы построения эффективных алгоритмов на языке ассемблера. Многочисленные примеры демонстрируют технику использования ассемблера для написания эффективного кода. Как и в остальных главах, основной упор делается на практическое применение языка. Необходимые теоретические сведения приводятся в контексте примеров и задач.

Предполагается, что читатель уже знаком с системой команд языка и знает в общих чертах, например, как выполняются *команды условных переходов*, *операции пересылки данных* или *битовые операции*. Примеры программного кода, приведенные в данной главе, могут без каких-либо изменений использоваться для решения собственных задач.

Как уже отмечалось, ассемблер чаще всего применяется для программирования математических алгоритмов, задач быстрой сортировки и поиска данных в массивах и для оптимизации циклически повторяющихся вычислений. Такие задачи очень часто решаются при разработке программ на языках высокого уровня и занимают значительное время программных вычислений.

Мы будем рассматривать только те команды ассемблера, которые являются общими для всех процессоров Intel, начиная с 386. В последних поколениях процессоров появились команды, позволяющие выполнять быструю обработку массивов данных, а также комплексные команды, позволяющие опти-

мизировать сам алгоритм вычислений. Это касается семейства процессоров Pentium, Pentium Pro, Pentium II, Pentium III.

Для демонстрации примеров используются только 32-разрядные вычисления. Это означает, что, например, при адресации регистров процессора будет использоваться мнемоника EAX, EBX, ECX и т. д. Все операции с памятью предполагают использование 32-разрядных операндов. Мы не будем ссылаться на *сегментные* регистры CS, DS и ES, т. к. в 32-разрядной модели вычислений отсутствует само понятие сегментных регистров. Более того, попытка использовать эти регистры для каких-либо операций сразу приведет к аварийному завершению программы.

Прежде чем начать программировать на ассемблере, необходимо определиться с инструментами разработки программ. Для демонстрации примеров программного кода, написанного на ассемблере, использован пакет MASM 6.15 фирмы Microsoft.

Хочется напомнить, что за пределами нашего внимания оказался удобный компилятор NASM, занимающий первое место в мире по популярности среди разработчиков и превосходящий по своим возможностям как ассемблер фирмы Microsoft, так и Турбо ассемблер Borland. Однако российский читатель ближе знаком с макро ассемблером Microsoft, поэтому будем ссылаться в дальнейшем именно на этот пакет.

В этой и во всех последующих главах в примерах программного кода будет использоваться упрощенный синтаксис для обозначения логических сегментов кода и данных. Так, например, сегмент данных будет обозначаться как `.data`, а сегмент кода — `.code`.

Вначале рассмотрим принципы построения подпрограмм на языке ассемблера. Этот материал является очень важным, и мы будем опираться на него при рассмотрении других тем книги.

В дальнейшем термины "подпрограмма", "процедура" и "функция" будут использоваться как синонимы. Функция — это процедура, которая возвращает значение, поэтому, думается, не должно возникать путаницы в применении этих понятий.

2.1. Использование процедур в языке ассемблера

Необходимость в написании процедур возникает тогда, когда программа становится большой и сложной. В этом случае имеет смысл оформить в виде процедур повторяющиеся блоки кода.

Кратко напомним определение процедуры и способ ее вызова. Для обозначения начала и конца процедуры в языке ассемблера используются *директ-*

тивы proc и endp. Последней командой процедуры обычно является команда ret, а для вызова процедуры используется команда call. Фрагмент кода, приведенный в листинге 2.1, демонстрирует в общем виде принципы использования процедуры myProc.

Листинг 2.1. Вызов процедуры myProc из программы на ассемблере

```
  ...
.data
  ...
.code
start:
   ...
 call    myProc
   ...
myProc proc
   ...
  ret
myProc endp
end start
```

В процедуру обычно передаются один или несколько *параметров* или *аргументов*. Для 32-разрядных приложений любой параметр представлен *двойным словом* DWORD.

Общепринято, что процедура возвращает значение в регистре EAX (напомним, что мы рассматриваем только 32-разрядные приложения).

Параметры в вызывающую процедуру могут передаваться одним из нескольких способов. Наиболее распространенным из них является передача параметров через стек. Обычно такой способ используется для интерфейса программ на языке ассемблера с программами на языке высокого уровня. Далее показан фрагмент программного кода (листинг 2.2), демонстрирующий этот метод передачи параметров.

Листинг 2.2. Передача параметров в процедуру через стек

```
   ...
.data
  SUM      DD  0
  I1       DD  32
  I2       DD -43
.code
```

```
start:
   ...
  push    DWORD PTR I1
  push    DWORD PTR I2
  call    SumTwo
  mov     SUM, EAX
   ...
SumTwo proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, DWORD PTR [EBP+12]
  add     EAX, DWORD PTR [EBP+8]
  pop     EBP
  ret
SumTwo endp
  ...
end start
```

Перед вызовом процедуры SumTwo передаваемые параметры (два целых числа I1 и I2) помещаются в стек. I1 размещается в стеке по адресу [EBP+12], а I2 — по адресу [EBP+8]. Следует сказать, что стек увеличивается вниз, т. е. от верхних адресов памяти к нижним. Поэтому первый параметр, который помещается в стек, имеет больший адрес, чем второй.

Как только управление передается процедуре, параметры должны быть извлечены из стека для дальнейшей обработки. Лучше всего это можно сделать с помощью регистра EBP. Следующие две строки исходного текста демонстрируют, как это делается:

```
push    EBP
mov     EBP, ESP
```

После выполнения этих команд *фрейм* (окно) стека выглядит так, как показано на рис. 2.1.

Если необходимо сохранить регистры при вызове подпрограммы, то следует учитывать соответствующее смещение *указателя* стека. Большинство Windows-приложений должны сохранять в стеке регистры EBX, ESI и EDI. Тогда исходный текст процедуры SumTwo выглядел бы так, как показано в листинге 2.3.

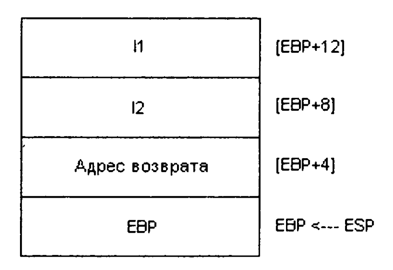


Рис. 2.1. Вид фрейма стека при вызове функции SumTwo

Листинг 2.3. Процедура SumTwo с сохранением регистров

```
SumTwo proc
  push    EBX
  push    ESI
  push    EBP
  mov     EBP, ESP
  mov     EAX, DWORD PTR [EBP+20]
  add     EAX, DWORD PTR [EBP+16]
  pop     EBP
  pop     ESI
  pop     EBX
  ret
SumTwo endp
```

В этом случае расположение элементов в стеке при вызове процедуры было бы как на рис. 2.2.

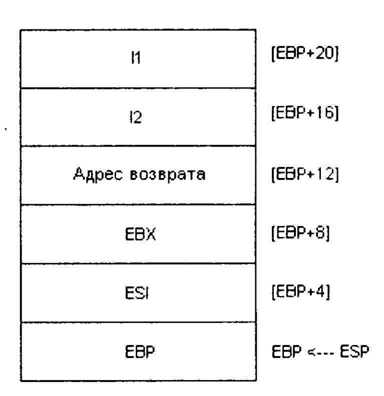


Рис. 2.2. Вид фрейма стека при сохранении в нем дополнительных регистров

На этом анализ работы нашей процедуры можно было бы закончить, если бы не одно "но". Приложение, использующее процедуру SumTwo, при выполнении завершится аварийно. В чем здесь дело? Еще раз внимательно проанализируем исходный текст фрагмента, где используется наша процедура:

```
push    DWORD PTR I1
push    DWORD PTR I2
call    SumTwo
mov     SUM, EAX
```

После завершения выполнения SumTwo в стеке остаются два значения переменных I1 и I2. При выходе из процедуры мы не восстановили стек, что в 100% случаев приводит к краху программы. Очистить стек можно одним из двух способов. Первый заключается в том, чтобы использовать следующую команду:

```
add     ESP, 8
```

Второй способ — использовать команду ret 8. Первый способ обычно используется вызывающей программой, второй — вызываемой процедурой. Далее показаны исправленные фрагменты программного кода для обоих вариантов.

1. Восстановление стека вызывающей программой:

```
  ...
push    DWORD PTR I1
push    DWORD PTR I2
call    SumTwo
mov     SUM, EAX
add     ESP, 8
  ...
```

2. Восстановление стека вызываемой процедурой:

```
SumTwo proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, DWORD PTR [EBP+12]
  add     EAX, DWORD PTR [EBP+8]
  pop     EBP
  ret     8
SumTwo endp
```

Большинство языков высокого уровня используют описанную методику для работы с внешними процедурами, и мы вернемся к этой теме при рассмотрении интерфейсов ассемблерных подпрограмм с языками высокого уровня.

Параметры в вызываемую процедуру могут передаваться не только через стек, но и через регистры. Покажем, как можно модифицировать предыдущий пример, если использовать для передачи параметров регистры EBX и ECX. Фрагмент кода вызывающей программы представлен в листинге 2.4.

Листинг 2.4. Передача параметров в процедуру через регистры

```
  ...
.data
 SUM   DD 0
 I1    DD 32
 I2    DD -43
.code
start:
  ...
 push  EBX
 push  ECX
 mov   EBX, DWORD PTR I1
 mov   ECX, DWORD PTR I2
 call  SumTwo
 mov   SUM, EAX
 pop   ECX
 pop   EBX
  ...
```

Исходный текст программного кода процедуры SumTwo (листинг 2.5) также изменится.

Листинг 2.5. Процедура SumTwo при передаче параметров через регистр

```
SumTwo proc
 mov  EAX, EBX
 add  EAX, ECX
 ret
SumTwo endp
```

Как видите, в этом примере передача параметров через стек обладает преимуществом перед регистровым методом. Это обусловлено тем, что вызы-

2 Зак. 1064

вающая программа практически всегда должна сохранять в стеке содержимое регистров, используемых при вызове процедуры. Производительность программы, использующей регистры для передачи параметров, может несколько снизиться, особенно при циклически выполняемых вычислениях.

Параметры процедур в наших примерах представляют собой значения переменных. Однако в большинстве случаев программисты передают параметры через указатели. Указатель на переменную представляет собой адрес, по которому эта переменная размещена в памяти. Как и переменные, указатели представляют собой 32-битовые величины в Windows. Использование указателей значительно упрощает работу с массивами переменных и во многих случаях является более эффективным, чем передача в процедуру значений переменных.

Процедуру `SumTwo` можно легко переделать для использования указателей вместо переменных. При передаче параметров через стек исходный текст процедуры будет выглядеть так, как показано в листинге 2.6.

Листинг 2.6. Обработка параметров-указателей в процедуре `SumTwo`

```
SumTwo proc
 push  EBX
 push  EBP
 mov   EBP, ESP
 mov   EAX, DWORD PTR [EBP+12]
 mov   EBX, DWORD PTR [EBP+16]
 mov   EAX, [EAX]
 add   EAX, [EBX]
 pop   EBP
 pop   EBX
 ret   8
SumTwo endp
```

Фрагмент кода основной программы для вызова процедуры `SumTwo` тоже изменится (листинг 2.7).

Листинг 2.7. Передача указателей в процедуру через стек

```
  ...
.data
 SUM   DD 0
 I1    DD 32
```

```
 I2    DD -43
.code
start:
  ...
 push offset I1
 push offset I2
 call SumTwo
 mov  SUM, EAX
  ...
```

При анализе исходного текста процедуры SumTwo видно, что для извлечения операндов требуются дополнительные команды. Кроме того, в процедуре, использующей указатели, как правило, необходимо задействовать дополнительные регистры, сохранив перед этим их содержимое в стеке.

Здесь не рассматривается способ передачи параметров с использованием общих областей памяти. Этот способ используется только MS-DOS приложениями и прямых аналогов в Windows не имеет. Вместо этого в операционных системах Windows для межпрограммных взаимодействий используется метод отображения файлов в память. Это отдельная тема и в книге не рассматривается. Но и полученных сведений будет вполне достаточно как для разработки своих подпрограмм, так и для понимания принципов работы чужих.

Полнофункциональные графические приложения Windows будут рассмотрены в последующих главах, а уже сейчас хотелось бы тестировать приложения на ассемблере. Простейший способ продемонстрировать работу программы — вывести результаты на дисплей. Это может выполнить консольное приложение.

В "старой доброй" MS-DOS для иллюстрации работы приложений использовалось обычное окно командного процессора, часто называемое DOS-окном. Создавать и отображать такое окно было легко, для этого достаточно было вызвать одну из системных функций, выводящую что-либо на экран.

Для того чтобы сделать то же самое в Windows, нам придется, несколько опережая события, познакомиться с принципом работы консольных приложений в этой операционной среде.

Подробное рассмотрение консольных приложений представлено в *главе 5*, сейчас же ограничимся только теми сведениями, которые необходимы, чтобы в общих чертах понять работу программного кода. Программный код каркаса или шаблона такого приложения довольно прост по сравнению с полнофункциональными графическими приложениями Windows. Мы разработаем два шаблона классического консольного приложения, которые и будем использовать для демонстрации примеров этой главы. Даже если что-то и будет непонятно в исходном тексте программ консоли, можно пропустить описание, тем более что это не помешает анализу примеров.

Один из шаблонов консольного приложения написан целиком на ассемблере и использует функции интерфейса прикладного программирования API операционной системы Windows. Обработка данных в такой программе выполняется при помощи команд ассемблера, а преобразование результата такой обработки в формат, удобный для отображения на экране дисплея, выполняется функциями WIN API. Такие приложения мы будем использовать в основном, для работы с целочисленными и строковыми переменными.

Второй шаблон приложения консоли мы разработаем с использованием мастера приложений Delphi 7. Каркас программного кода при использовании такого метода получается очень простым и будет понятен всем программистам без исключения: как поклонникам ассемблера, так и сторонникам языка C. В таком приложении вычислительные алгоритмы на языке ассемблера реализуются в виде процедур в теле основной программы. Это позволяет без ограничения общности работать с такими процедурами так же, как и с обычными ассемблерными программами. Консольные приложения на Delphi позволяют более гибко выполнять операции преобразования и ввода-вывода любых данных числового или текстового типа.

Мы будем рассматривать оба типа консольных приложений для отображения результатов работы наших демонстрационных программ.

Разработаем первый вариант консольного приложения, где будем использовать функции WIN API. Консольное приложение Windows представляет собой один из классов 32-разрядных приложений, в которых используется оконный интерфейс для работы в текстовом режиме. По внешнему виду консольное окно Windows очень сильно напоминает окно приложения MS-DOS. Проведем краткий сравнительный анализ приложений, работающих в MS-DOS и в Windows. Возьмем классический вариант консольного приложения — программу, выводящую строки `"Привет, мир!"`. В MS-DOS текст 16-разрядного приложения выглядел бы так, как показано в листинге 2.8.

Листинг 2.8. Консольное приложение MS-DOS

```
.model  small
.stack  100h
.data
  message   DB "Привет, мир!", 0
  lmessage  EQU $-message
.code
main proc
  mov     AX, @data
  mov     DS, AX
  mov     AH, 40h
```

```
  mov     DX, offset message
  mov     HX, 1
  mov     CX, lmessage
  int     21h
  mov     AX, 4C00h
  int     21h
main endp
end
```

Исходный текст программы консольного приложения Windows выглядит несколько иначе. Сразу же оговорюсь: мы будем использовать упрощенный синтаксис для компилятора MASM. Исходный текст программы для компилятора Microsoft представлен в листинге 2.9.

Листинг 2.9. Консольное приложение Windows

```
.386
.model flat, stdcall
  option casemap : none                    ; различаем регистр символов

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib

.data

  ; объявления и инициализация переменных

  conTitle       DB  "Assembler console application", 0
  mes            DB  "Привет, мир!", 0
  len_mes        EQU $-mes
  readBuf        DB  ?
  lenReadBuf     DD  1
  hStdIn         DD  0
  hStdOut        DD  0
  chrsRead       DD  0
```

```
  chrsWritten   DD  0

  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11

.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex

  ; инициализация консольного приложения

  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex
  call    getout_hndl
  call    getinp_hndl

  ; вывод сообщения в окно консоли

  push    EBX
  mov     EBX, offset mes
  mov     ECX, len_mes
  call    write_con
  pop     EBX

  ; ожидание ввода и выход из программы

  call    read_con
ex:
  push    0
  call    ExitProcess

;------ Процедуры -----
getout_hndl proc
```

```
  push    STD_OUTP_HNDL
  call    GetStdHandle
  mov     hStdOut, EAX
  ret
getout_hndl endp

getinp_hndl proc
  push    STD_INP_HNDL
  call    GetStdHandle
  mov     hStdIn, EAX
  ret
getinp_hndl endp

write_con proc
  push    0
  push    chrsWritten
  push    ECX
  push    EBX
  push    hStdOut
  call    WriteConsoleA
  ret
write_con endp

read_con proc
  push    0
  push    chrsRead
  push    lenReadBuf
  push    offset readBuf
  push    hStdIn
  call    ReadConsoleA
  ret
read_con endp

end start
```

Как видим, тексты двух программ сильно различаются. Тем не менее оба приложения используют одни и те же принципы. И приложение MS-DOS, и Windows-приложение используют механизм системных вызовов.

Программа MS-DOS (см. листинг 2.8) использует стандартный системный вызов через *прерывание* 21h для записи строки на консоль. При вызове используются параметры:

```
mov     AH, 40h              ; дескриптор вызова (AH=40h-запись
                             ; в файл-устройство)
mov     DX, offset message   ; адрес строки для записи
mov     BX, 1                ; дескриптор стандартного устройства вывода
mov     CX, lmessage         ; длина строки данных
int     21h
```

В операционных системах Windows системные вызовы реализованы иначе. Вместо прерывания 21h используется набор многочисленных функций WIN API, предоставляемых операционной системой Windows в распоряжение программиста. Механизм действия этих функций довольно сложный и будет рассмотрен в *главе 4* и в последующих главах. Сейчас же вполне достаточно знать несколько основных правил при работе с WIN API:

- параметры функций являются 32-разрядными;
- параметры функций передаются через стек;
- параметры записываются в стек в порядке, обратном их следованию в описании функции;
- если функция возвращает значение, то оно передается в регистре EAX.

Рассмотрим работу консольного приложения (см. листинг 2.9) более подробно. Для удобства фрагменты программного кода, выполняющие инициализацию, ввод и вывод, реализованы в виде отдельных подпрограмм.

Вначале приложение запрашивает у операционной системы текстовое окно (консоль) для вывода информации. Для этого используется функция WIN API AllocConsole. Она не требует никаких параметров при вызове. В случае успеха возвращается ненулевое значение, и программа продолжает работу. Если запрос на открытие консоли завершился неудачей, функция возвращает нулевое значение, и программа завершается.

```
  call    AllocConsole
  test    EAX, EAX
  jz      ex                  ; завершить работу
   ...
ex:
  push    0
  call    ExitProcess
```

В этом фрагменте программного кода мы столкнулись еще с одной функцией WIN API — `ExitProcess`. Этой функцией приложение, как правило, завершает работу. В качестве входного параметра задаем нулевое значение. Функция `ExitProcess` является аналогом системного вызова MS-DOS, завершающего работу DOS-приложения:

```
mov    AX, 4C00h
int    21h
```

Предположим, мы успешно запросили у Windows консоль. Назовем окно консоли, например, `"Assembler console application"`, вызвав для этого функцию WIN API `SetConsoleTitleA`. В качестве параметра эта функция принимает смещение строки с заголовком.

```
push  offset conTitle
call  SetConsoleTitleA
test  EAX, EAX
jz    ex
```

Как и при вызове функции `AllocConsole`, в регистре `EAX` возвращается флаг успешного или неудачного выполнения функции `SetConsoleTitleA`. Если и этот вызов закончился успешно, то можно выводить строку `"Привет, мир!"` в окно приложения.

Вывод осуществляется функцией `WriteConsoleA`, которая записывает строку символов на экран с текущей позиции *курсора*. Функция объявляется следующим образом:

```
BOOL WriteConsoleA(HANDLE  hConsoleOutput,
                   CONST   VOID *lpBuffer,
                   DWORD   nNumberOfCharsToWrite,
                   LPDWORD lpNumberOfCharsWritten,
                   LPVOID  lpReserved)
```

Параметры функции `WriteConsoleA`:

- `HANDLE hConsoleOutput` — дескриптор (*идентификатор*) выходных данных консоли;
- `CONST VOID *lpBuffer` — указатель на *буфер* выводимой строки;
- `DWORD nNumberOfCharsToWrite` — количество выводимых символов;

- LPDWORD lpNumberOfCharsWritten — указатель на фактическое количество выведенных символов;
- LPVOID lpReserved — зарезервировано (должно быть равно 0).

Функция возвращает ненулевое значение в случае успешного завершения.

Читателя могут смутить идентификаторы в описании параметров функции — HANDLE, LPDWORD, LPVOID. Несмотря на столь загадочную аббревиатуру, используемую Microsoft, эти параметры имеют простой смысл. HANDLE представляет собой переменную целого типа размером в двойное слово DWORD, а параметры с префиксом LP (LPDWORD, LPVOID) — указатели на переменную, также размером в двойное слово.

При вызове функций WIN API параметры передаются через стек справа налево, причем первым попадает в стек самый правый параметр.

```
push    0
push    chrsWritten
push    len_charBuf
push    offset charBuf
push    hStdOut
call    WriteConsoleA
```

Для удобства использования этой функции в последующих примерах можно поместить ее исходный код в подпрограмму write_con.

```
write_con proc
  push    0
  push    chrsWritten
  push    ECX
  push    EBX
  push    hStdOut
  call    WriteConsoleA
  ret
write_con endp
```

Параметры для вызова этой функции будем передавать в регистрах ECX и EBX. Содержимое регистра EBX желательно сохранить в стеке. Пример вызова функции write_con из нашего приложения будет выглядеть следующим образом:

```
push    EBX
mov     EBX, offset charBuf
```

```
mov     ECX, len_charBuf
call    write_con
pop     EBX
```

Другой полезной функцией WIN API является `ReadConsoleA`, вызываемая из процедуры `read_con`.

```
read_con proc
  push    0
  push    chrsRead
  push    lenReadBuf
  push    offset readBuf
  push    hStdIn
  call    ReadConsoleA
  ret
read_con endp
```

Функция `ReadConsoleA` считывает символ из буфера ввода консоли и очищает буфер. В случае успеха функция возвращает ненулевое значение.

Мы используем ввод с консоли в программе для задержки закрытия окна приложения после выводастроки на экран.

Функция объявляется следующим образом:

```
BOOL ReadConsoleA(HANDLE  hConsoleInput,
                  LPVOID  lpBuffer,
                  DWORD   nNumberOfCharsToRead,
                  LPDWORD lpNumberOfCharsRead,
                  LPVOID  lpReserved);
```

В функцию передаются следующие параметры:

- `HANDLE hConsoleInput` — дескриптор буфера входных данных;
- `LPVOID lpBuffer` — буфер данных;
- `DWORD nNumberOfCharsToRead` — количество символов для чтения;
- `LPDWORDlpNumberOfCharsRead` — количество символов, фактически прочитанных;
- `LPVOIDlpReserved` — зарезервировано (обычно равно 0).

Программа выводит строку "Привет, мир!" в окно консольного приложения и ожидает ввода символа с клавиатуры, после чего ее работа завершается. Чтобы использовать каркас приложения для демонстрации наших примеров, будем размещать программный код между секцией инициализации и функцией чтения консоли. Компиляция собранного таким образом приложения выполняется с помощью командных строк:

```
ml/c/coff/Fo hellow.obj hellow.asm
link/SUBSYSTEM:WINDOWS/LIBPATH: <disk>:\masm\lib hellow.obj
```

Второй вариант консольного приложения Windows разработаем с использованием Мастера приложений Delphi 7. Конечно, читатели легко могут выполнить все необходимые манипуляции в среде программирования Delphi для получения каркаса такого приложения, поэтому не будем останавливаться на этом подробно.

Полученный таким образом шаблон 32-разрядного консольного приложения представлен в листинге 2.10.

Листинг 2.10. Шаблон 32-разрядного консольного приложения, созданного с помощью Мастера приложений Delphi 7

```
program generic;

{$APPTYPE CONSOLE}

uses
  SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Как использовать язык ассемблера в таком приложении, не ограничивая его возможности рамками среды Delphi? В Delphi предусмотрена замечательная возможность разрабатывать и использовать процедуры, целиком написанные на ассемблере, прямо в основной программе так, как будто бы они были отдельными объектными модулями. Процедура, написанная целиком на ассемблере, объявляется специальной директивой assembler и может выглядеть так, как показано в листинге 2.11.

Листинг 2.11. Процедура, целиком написанная на ассемблере

```
function SumTwo(I1: Integer; I2: Integer): Integer; assembler;
asm
  mov     EAX, DWORD PTR I1
  add     EAX, DWORD PTR I2
end;
```

В этой простой процедуре выполняется операция сложения двух целых чисел и результат, как обычно, возвращается в регистре EAX. Поскольку целочисленные переменные типа `Integer` имеют размер двойного слова, то смысл операторов процедуры становится понятным. Теперь можно собрать консольное приложение (листинг 2.12), выполняющее вывод на экран суммы двух чисел.

Листинг 2.12. Консольное приложение, выполняющее суммирование двух чисел

```
program generic;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  I1, I2, ISUM: Integer;

function SumTwo(I1: Integer; I2: Integer): Integer; assembler;
asm
  mov     EAX, DWORD PTR I1
  add     EAX, DWORD PTR I2
end;

begin
  { TODO -oUser -cConsole Main : Insert code here }

  I1 := -23;
  I2 := -56;
  ISUM := SumTwo(I1, I2);
  WriteLn(ISUM);
  ReadLn;
end.
```

Исходный текст приложения понятен для опытных программистов и не требует дополнительных комментариев. Переделка подобной программы для компиляции в C++ потребует всего несколько минут. И еще одно замечание. Все демонстрационные программы этой главы будут использовать только вывод данных на экран дисплея. Чтобы избежать лишней сложности примеров, мы не будем рассматривать в этой главе операции, связанные с файловым вводом-выводом, манипуляции с памятью и процессами.

Рассмотрим далее наиболее полезные алгоритмы и процедуры на языке ассемблера. Начнем с математических вычислений. Практически все приложения используют те или иные операции, связанные с математическими вычислениями, начиная от простейших (сложение и вычитание) и заканчивая решением систем уравнений. Математические операции могут использовать как обычные команды процессора, например `add`, `sub`, `mul` и `div`, так и специальные команды математического сопроцессора.

2.2. Реализация математических вычислений на языке ассемблера

Арифметические команды любого микропроцессора привлекают к себе наибольшее внимание. Хотя их немного, они выполняют основное количество преобразований данных в процессоре. В реальных же условиях арифметические команды занимают лишь малую часть всех исполняемых команд, но имеют определенную специфику выполнения. Языки высокого уровня отличаются многообразием и мощью своих математических функций. Однако в основе математических библиотек языков высокого уровня лежит относительно простой набор команд основного процессора и сопроцессора. В языке ассемблера нет таких комплексных функций и библиотек, однако можно разработать свои процедуры, ничем не уступающие функциям языков высокого уровня. Для этого нужно лишь использовать те возможности, которые предоставили нам разработчики фирмы Intel.

Начнем с рассмотрения арифметических команд основного процессора, а именно с команды сложения `add`. Команда `add` выполняет сложение указанных операндов, представленных в двоичном дополнительном коде. Результат помещается в первый операнд, а второй операнд не изменяется. Команда корректирует *регистр флагов* в соответствии с результатом сложения. Например, вызов

```
add     EAX, EBX
```

суммирует содержимое регистров EAX и EBX, а результат помещает в регистр EAX. Биты регистра флагов устанавливаются в соответствии с тем, был ли результат нулевым, отрицательным, имел ли *четность*, *перенос* или *переполнение*. Операция сложения выполняется только для однотипных операндов.

В качестве операндов могут выступать регистры, ячейки памяти и непосредственные операнды, однако нельзя складывать два операнда в ячейках памяти.

Команда сложения с переносом adc — это модификация команды add, за исключением того, что в сумму включается *флаг переноса*. Обе команды — add и adc — устанавливают в положение 1 флаг переноса, если произошел перенос из старшего разряда результата. Команда add складывает два операнда без учета флага переноса, а команда adc учитывает флаг переноса. При установленном в 0 флаге переноса результат выполнения adc совпадает с результатом выполнения команды add. Если же флаг переноса установлен в положение 1, то результат выполнения команды adc будет больше на единицу результата команды add. Таким образом, программа может использовать флаг переноса для выполнения операций *повышенной точности*. Приведем листинг программы сложения для двух чисел с повышенной точностью (листинг 2.13).

Листинг 2.13. Программа сложения двух чисел с повышенной точностью

```
.386
.model flat, stdcall
  option casemap :none                        ; различаем регистр символов

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib

.data

  ; объявления и инициализация переменных

  conTitle      DB  "Adding of two integers", 0
  mes           DB  "The result of adding -8709 and 3657 = ", 0
  len_mes       EQU $-mes
  charBuf       DB  "       ", 0
  len_charBuf   DD  $-charBuf
  i1            DD -8709
  i2            DD  3657
```

```
  lpFmt         DB  "%d", 0
  readBuf       DB  ?
  lenReadBuf    DD  1
  hStdIn        DD  0
  hStdOut       DD  0
  chrsRead      DD  0
  chrsWritten   DD  0

  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11

.code
start:
  call    AllocConsole
  test    EAX, EAX

  jz      ex

  ; инициализация консольного приложения

  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex
  call    getout_hndl
  call    getinp_hndl

  ; сложение двух двойных слов

  mov     AX, WORD PTR i1
  add     WORD PTR i2, AX             ; сложение младших 16 разрядов
  mov     AX, WORD PTR i1+2           ; сложение старших 16 разрядов
  adc     WORD PTR i2+2, AX

  ; преобразование результата сложения в строку

  push    DWORD PTR i2
  push    offset lpFmt
```

```
  push    offset charBuf
  call    wsprintf
  add     ESP, 12

  ; вывод сообщения в окно консоли

  push    EBX
  mov     EBX, offset mes
  mov     ECX, len_mes
  call    write_con
  pop     EBX

  ; вывод результата сложения в окно консоли

  push    EBX
  mov     EBX, offset charBuf
  mov     ECX, len_charBuf
  call    write_con
  pop     EBX

  ; ожидание ввода и выход из программы

  call    read_con
ex:
  push    0
  call    ExitProcess

;------ Процедуры ----

getout_hndl proc
  push    STD_OUTP_HNDL
  call    GetStdHandle
  mov     hStdOut, EAX
  ret
getout_hndl endp

getinp_hndl proc
  push    STD_INP_HNDL
```

```
  call    GetStdHandle
  mov     hStdIn, EAX
  ret
getinp_hndl endp

write_con proc
  push    0
  push    chrsWritten
  push    ECX
  push    EBX
  push    hStdOut
  call    WriteConsoleA
  ret
write_con endp

read_con proc
  push    0
  push    chrsRead
  push    lenReadBuf
  push    offset readBuf
  push    hStdIn
  call    ReadConsoleA
  ret
read_con endp

end start
```

Операция суммирования в этой программе выполняется последовательностью следующих команд:

```
mov     AX, WORD PTR i1
add     WORD PTR i2, AX
mov     AX, WORD PTR i1+2
adc     WORD PTR i2+2, AX
```

Как видно из этого фрагмента кода, операция сложения двух целочисленных 32-битовых операндов разбита на два этапа. Вначале находится сумма двух младших слов обоих операндов, затем двух старших. К полученной сумме добавляется содержимое флага переноса, а результат помещается

в переменную i2. Необходимо также поместить один из операндов в регистр. Первое сложение выполняется командой add, т. к. текущее значение флага переноса для него несущественно. После этого выполняется второе сложение командой adc с учетом флага переноса, установленного предыдущим сложением.

Для вывода полученного результата в окно консоли необходимо вначале преобразовать целое число в строку символов. Такое преобразование можно выполнить при помощи функции WIN API wsprintf. Эта функция очень полезна, т. к. позволяет преобразовать данные арифметического типа в строку. Функция wsprintf будет использоваться нами чрезвычайно широко в примерах последующих глав, поэтому остановимся на ней более подробно.

Функция wsprintf форматирует и запоминает строки символов и арифметических операндов в буфере. Любые аргументы конвертируются и запоминаются в буфере в соответствии с определенными для них спецификациями форматирования. Функция формирует в буфере строку с завершающим нулем и возвращает размер строки в качестве результата. Функция wsprintf объявляется следующим образом:

```
int wsprintf(LPTSTRlpOut, LPCTSTRlpFmt,...);
```

где параметры функции:

- LPTSTRlpOut — буфер вывода;
- LPCTSTRlpFmt — строка, задающая формат выводимых переменных.

Третьим параметром функции wsprintf является переменная или список переменных, которые необходимо преобразовать. Иными словами, функция wsprintf может принимать переменное число параметров.

Вызов этой функции в нашей программе и передаваемые параметры показаны в следующем фрагменте кода:

```
  ...
charBuf         DB "        ", 0
len_charBuf     DD $-charBuf
lpFmt           DB "%d", 0
i2              DD 3657
  ...
push    DWORD PTR i2
push    offset lpFmt
push    offset charBuf
call    wsprintf
add     ESP, 12
  ...
```

Параметры передаются, как обычно, через стек, справа налево. Но есть один нюанс, связанный с так называемым соглашением о передаче параметров. Все функции WIN API используют соглашение `stdcall`. Это значит, что по завершению выполнения вызываемой процедуры она сама очищает стек. Однако функция `wsprintf` использует другое соглашение — `cdecl`, а это означает, что очищать стек должна вызывающая процедура, т. е. наша программа. Поэтому в исходный текст за оператором вызова процедуры необходимо вставить следующую строку:

```
add    ESP, 12,
```

Функция `wsprintf` принимает 3 параметра (это 12 байт), а поскольку стек растет к меньшим адресам, то смысл этого оператора становится понятным. Мы подробно остановимся на вопросах, касающихся передачи параметров из языков высокого уровня в ассемблерные подпрограммы, в *главе 3*.

Можно рекомендовать программистам, пишущим на ассемблере, использовать `wsprintf` и другие функции преобразования данных, входящие в интерфейс WIN API, в своих приложениях. Можно, конечно, разработать свои собственные процедуры преобразования, например из числового формата в строку и наоборот, однако в этом нет особого смысла. Процедуры преобразования различных типов переменных, написанные разработчиками Microsoft, хорошо оптимизированы, и их применение значительно сэкономит время.

Сохраним текст нашей программы в файле с расширением ASM и откомпилируем ее. Окно работающего приложения показано на рис. 2.3.

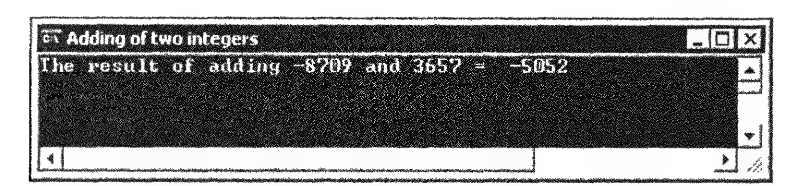


Рис. 2.3. Окно приложения, выполняющего суммирование двух чисел

Мы так подробно рассмотрели наши первые приложения для того, чтобы в последующих примерах главы не акцентировать внимание на интерфейсе с операционной системой Windows, а сосредоточиться на анализе интересующего нас кода.

Рассмотрим теперь вычитание двух операндов с повышенной точностью. Для этой операции будем использовать команды `sub` и `sbb`.

Команды вычитания `sub` и `sbb` являются как бы зеркальным отображением команд сложения. Команды устанавливают флаги состояния в соответствии

с результатом операции. Флаг переноса будет означать заем единицы. Например, команда

```
sub     EAX, EBX
```

выполнит вычитание содержимого регистра EBX из содержимого регистра EAX, поместив результат в EAX. Флаги состояния будут установлены в соответствии с результатом выполнения команды.

Вычитание с заемом выполняется с помощью команды sbb. Эта команда используется в операциях вычитания с повышенной точностью. При выполнении команды sbb необходимо учитывать значение *флага заема*. Для получения правильного результата значение заема вычитается из результата, полученного при нормальном вычитании.

Для демонстрации вычитания с повышенной точностью воспользуемся исходным текстом предыдущего примера, сделав некоторые изменения. Заменим группу команд сложения

```
mov     AX, WORD PTR i1
add     WORD PTR i2, AX
mov     AX, WORD PTR i1+2
adc     WORD PTR i2+2, AX
```

на группу команд вычитания

```
mov     AX, WORD PTR i1
sub     WORD PTR i2, AX
mov     AX, WORD PTR i1+2
sbb     WORD PTR i2+2, AX
```

Для разнообразия поменяем значения операндов i1 и i2 следующим образом:

```
i1      DD 18709
i2      DD -9657
```

Заголовок окна приложения также изменим:

```
conTitle      DB "Substraction of two integers", 0
```

Сохраним наш исходный текст в файле с расширением ASM и откомпилируем приложение. После запуска можно видеть результат работы программы на рис. 2.4.

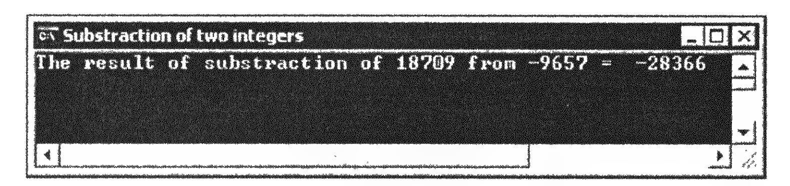


Рис. 2.4. Окно приложения, выполняющего вычитание двух чисел

Операции сложения и вычитания с повышенной точностью можно выполнять и для многобайтных операндов. В предыдущих двух примерах мы выполняли действия с целыми числами, которые в большинстве систем программирования представлены как двойное слово (4 байта). Однако не следует полагаться на то, что переменные целого типа обязательно будут иметь размерность в 4 байта! Поскольку многие ассемблерные программы работают совместно с языками высокого уровня, необходимо четко представлять, какие типы данных и какой размерности использует компилятор C++ .NET или Delphi.

В компиляторе Visual C++ .NET, например, существует несколько типов переменных целого типа, имеющих различную размерность. При работе с этим компилятором с уверенностью можно полагаться только на тип `_intX`, где `X` может принимать значения 8, 16, 32, 64. Это так называемый тип целочисленных данных с фиксированной размерностью.

Кроме того, приложения на ассемблере широко используют функции WIN API, которые оперируют разными типами данных. Необходимо внимательно относится к использованию переменных при вызове этих функций. Неправильный формат данных (не только целочисленных) приводит к трудно обнаруживаемым ошибкам.

В программистской практике редко бывает так, чтобы все числовые величины имели одинаковую размерность. В рассмотренном выше примере сложения двух целых чисел оба операнда имели размерность двойного слова. Предположим, что один из этих операндов является байтом. Как в этом случае выполнить операцию сложения, которая требует два операнда одинаковой размерности? В этих случаях используются так называемые команды преобразования типа. При помощи таких команд можно расширить байт до слова, а слово — до двойного слова. Двойные слова можно расширить до учетверенных слов. К командам преобразования типов относятся:

- `cbw` — команда преобразования байта в регистре `AL` в слово в регистре `AX`. При таком преобразовании старший бит в регистре `AL` расширяется на все биты регистра `AH`;

- cwd — команда преобразования слова в регистре AX в двойное слово в регистрах DX:AX. В этом случае старший бит регистра AX расширяется на биты регистра DX;
- cwde — команда преобразования слова в регистре AX в двойное слово в регистре EAX через расширение старшего бита AX на старшие 16 бит регистра EAX;
- cdq — команда преобразования двойного слова в регистре EAX в учетверенное слово в регистрах EDX:EAX через расширение старшего бита в регистре EAX на все биты EDX.

Изменим пример для нахождения суммы двух целых чисел, предположив, что один из операндов представляет собой байт. Фрагменты кода, которые необходимо вставить в исходный текст программы, приведены в листинге 2.14.

Листинг 2.14. Фрагменты кода, выполняющие сложение двух операндов, один из которых — байт

```
 ...
.data
 conTitle     DB  "Assembler console application", 0
 mes          DB  "The result of adding 3600 and -18 = ", 0
 en_mes       EQU  $-mes
 charBuf      DB  "       ", 0
 len_charBuf DD  $-charBuf
 ib1          DB  -18
 id1          DD  0
 id2          DD  3600
 lpFmt        DB  "%d", 0
  ...
  ...
 mov     AL, ib1
 cbw
 cwde
 mov     id1, EAX
 mov     AX, WORD PTR id1
 add     WORD PTR id2, AX
 mov     AX, WORD PTR id1+2
 adc     WORD PTR id2+2, AX
  ...
```

Одним из операндов в модифицированной программе является целочисленная переменная ib1 размерностью в 1 байт. Как видно из фрагмента кода, преобразование выполняется в два этапа: сначала 1-байтовая переменная загружается в регистр AL, где преобразуется в слово, затем слово в регистре EAX преобразуется в двойное слово и помещается в регистр EAX. Далее мы помещаем переменную из регистра EAX в переменную id1. Затем выполняется сложение двух чисел в переменных id1 и id2 обычным способом.

Следующий пример демонстрирует, как найти сумму элементов целочисленного массива, используя операции сложения с повышенной точностью. Возьмем массив из 7 целых чисел и напишем программный код, в качестве каркаса используя пример суммирования двух целых чисел. Фрагмент программного кода приведен в листинге 2.15.

Листинг 2.15. Фрагмент программного кода, выполняющего суммирование элементов целочисленного массива

```
.386
.model flat, stdcall
  option casemap :none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib

.data
  conTitle     DB  "Sum of integers in array", 0
  mes          DB  "The result of summation of integers = ", 0
  len_mes      EQU $-mes
  charBuf      DB  "       ", 0
  len_charBuf  DD  $-charBuf
  iarray       DD -90, 34, -67, 32, 11, -5, 41
  len_iArray   EQU ($-iarray)/4
  ISUM         DD  0
  lpFmt        DB  "%d", 0
  readBuf      DB  ?
  lenReadBuf   DD  1
  hStdIn       DD  0
```

```
  hStdOut       DD  0
  chrsRead      DD  0
  chrsWritten   DD  0
  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11
.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex
  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex
  call    getout_hndl
  call    getinp_hndl

  ; вычисление суммы чисел, входящих в массив

  lea     ESI, iarray   ; помещаем адрес первого элемента массива в ESI
  mov     ECX, len_iArray            ; помещает размер массива в ECX
next:
  mov     AX, WORD PTR [ESI]
  add     WORD PTR ISUM, AX          ; сумма младших 16-ти разрядов
  mov     AX, WORD PTR [ESI+2]
  adc     WORD PTR ISUM+2, AX        ; сумма старших 16-ти разрядов
  add     ESI, 4                     ; адрес следующего элемент массива
  loop    next

  ; преобразование результата в строку

  push    DWORD PTR ISUM
  push    offset lpFmt
  push    offset charBuf
  call    wsprintf
  add     ESP, 12

  push    EBX
```

```
  mov     EBX, offset mes
  mov     ECX, len_mes
  call    write_con
  pop     EBX

; вывод результата в окно приложения

  push    EBX
  mov     EBX, offset charBuf
  mov     ECX, len_charBuf
  call    write_con
  pop     EBX

; ожидание нажатия клавиши и выход

  call    read_con
ex:
  push    0
  call    ExitProcess
  ...
```

В этом фрагменте кода для суммирования элементов массива `iarray` используется цикл. На каждой итерации в регистр `ESI` загружается адрес двойного слова. Суммирование элементов выполняется с использованием 16-разрядного регистра `AX`. Количество итераций определяется содержимым счетчика на регистре `ECX`. В регистр `ECX` загружается размер массива в двойных словах. Результат выполнения каждой итерации добавляется к сумме, полученной ранее, и хранится в переменной `ISUM`.

Преобразование целочисленной переменной `ISUM` в строку выполняется, как мы уже знаем, функцией WIN API `wsprintf`. Вывод результата в окно приложения выполняется, как обычно, функцией `write_con` (рис. 2.5).

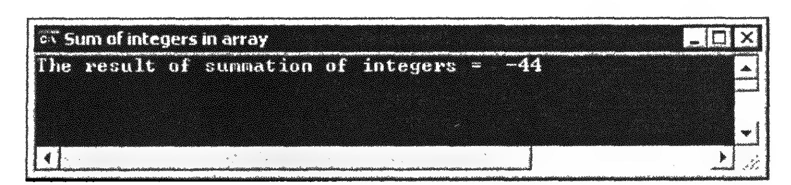


Рис. 2.5. Окно приложения, вычисляющего сумму элементов массива целых чисел

Следующими математическими операциями, которые мы рассмотрим, будут операции умножения и деления. Вначале расскажу коротко о том, как выполняются операции умножения. Существуют две команды умножения. Команда `mul` используется для умножения двух целых чисел без знака и дает беззнаковый результат. Команда `imul` используется для умножения целых чисел со знаком.

Команда `mul` выполняет умножение 8-, 16- или 32-разрядных операндов. Множимое помещается в регистр AL, AX или EAX, в зависимости от размера операнда. Множителем может быть 8-, 16- или 32-разрядное число. Размерности операндов множимого и множителя должны быть одинаковыми.

Размерность результата умножения в два раза больше размерности каждого из операндов. Так, если в операции умножения используется регистр AL, то результат помещается в регистр AX. Если используется регистр AX, то результат помещается в регистры DX:AX, причем, в регистре DX содержатся старшие 16 бит результата, а в регистре AX — младшие 16 бит. Наконец, если в качестве множимого используется регистр EAX, то результат помещается в регистры EDX:EAX. Старшие 32 бита результата помещаются в регистр EDX, младшие — в регистр EAX.

В операции умножения не допускается использование непосредственных операндов.

Следующий пример показывает, как выполняется операция умножения беззнаковых чисел в программе на ассемблере. В этой программе мы будем использовать 32-разрядные операнды. Особое внимание хочется обратить на технику обработки результата умножения в этом примере. Поскольку в результате умножения 32-битовых операндов мы получаем 64-битовый результат, то преобразование его в строковую переменную функцией `wsprintf` будет выглядеть сложнее, чем для 32-разрядных чисел. Исходный текст программы представлен в листинге 2.16.

Листинг 2.16. Программа умножения беззнаковых чисел

```
.386
.model flat, stdcall
  option casemap : none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib
```

```
.data
  conTitle      DB  "Multiplication of integers", 0
  mes           DB "The result of muliplying the ints = ", 0
  len_mes       EQU $-mes
  charBuf       DB  "       ", 0
  len_charBuf   DD  $-charBuf
  i1            DD  95
  i2            DD  34
  ires          DQ  0
  lpFmt         DB  "%ld", 0
  readBuf       DB  0
  lenReadBuf    DD  1
  hStdIn        DD  0
  hStdOut       DD  0
  chrsRead      DD  0
  chrsWritten   DD  0
  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11
.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex
  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex

  call    getout_hndl
  call    getinp_hndl

  ; умножение 32-разрядных операндов

  mov     EAX, i1
  mov     EDX, i2
  mul     EDX
  mov     DWORD PTR ires, EAX
  mov     DWORD PTR ires+4, EDX
```

```
; преобразование 64-разрядного целого в строку

push    DWORD PTR ires+4
push    DWORD PTR ires
push    offset lpFmt
push    offset charBuf
call    wsprintf
add     ESP, 16

push    EBX
mov     EBX, offset mes
mov     ECX, len_mes
call    write_con
pop     EBX

; вывод результата умножения на экран

push    EBX
mov     EBX, offset charBuf
mov     ECX, len_charBuf
call    write_con
pop     EBX

; ожидание ввода с клавиатуры и выход

call    read_con
ex:
push    0
call    ExitProcess
 ...

; объявления функций
 ...
end start
```

В этой программе выполняется умножение операндов, находящихся в 32-разрядных переменных `i1` и `i2`, а результат помещается в 64-битовую переменную `ires`, объявленную директивой `DQ` (учетверенное слово). Поскольку

результат операции умножения возвращается в двух регистрах EDX и EAX, то для его записи в ires используются команды

```
mov     DWORD PTR ires, EAX          ; младшая часть результата
mov     DWORD PTR ires+4, EDX        ; старшая часть результата
```

Для преобразования учетверенного слова в строку необходимо вызвать функцию wsprintf с четырьмя параметрами:

```
push    DWORD PTR ires+4
push    DWORD PTR ires
push    offset lpFmt
push    offset charBuf
call    wsprintf
add     ESP, 16
```

Напомню, что функция wsprintf может принимать переменное число параметров за счет того, что третий параметр имеет переменную длину. Так как параметры помещаются в стек в виде двойных слов, то для передачи параметра длиной в учетверенное слово потребуется два двойных слова, что и сделано в фрагменте кода. Соответственно, для восстановления стека необходимо удалять 16 байт с помощью команды:

```
add     ESP, 16
```

В зависимости от результата выполнения команды mul соответствующим образом устанавливаются флаг переноса CF и флаг переполнения OF. Если оба флага равны 0, то значение старшей части результата равно 0. Если флаг переноса установлен в положение 1, то результат размещается в обоих регистрах.

Для умножения чисел со знаком применяется команда imul. Выполняется она аналогично команде mul, единственным отличием является то, что формируется знаковый бит результата. Если флаг переноса и флаг переполнения равны 0, то содержимое старшего регистра результата является расширением знакового бита младшего регистра. Если оба флага устанавливаются в 1, то, в зависимости от разрядности результата, это означает следующее:

- 16-разрядный результат операции расширил знак в регистр AH;
- 32-разрядный результат расширил знак в регистр DX;
- 64-разрядный результат расширил знак в регистр EDX.

Операцией, противоположной умножению, является деление. Как и в случае умножения, существуют две команды деления — div (для двоичных чисел без знака) и idiv (для чисел со знаком). Любая из этих команд деления работает с байтами, словами и двойными словами.

Команда деления div выполняет деление 8-, 16- и 32-разрядных чисел без знака. В зависимости от размера делимого и делителя получаются следующие результаты:

- если делитель — 8-разрядное число, то делимое помещается в регистр AX, частное — в регистр AL, а остаток — в регистр AH;
- если делимое является 32-разрядным числом, то оно помещается в регистры DX:AX (в DX — старшая часть, в AX — младшая). Частное помещается в регистр AX, а остаток — в регистр DX;
- если делитель — 32-разрядное число, то делимое помещается в регистры EDX:EAX, частное — в регистр EAX, остаток — в регистр EDX.

Ни один из флагов состояния не определен после команды деления. Если частное больше, чем может быть помещено в регистр результата, результат будет неправильным. При делении байтов частное должно быть меньше 256, а при операции со словами — меньше 65 535. Процессор не устанавливает никаких флагов, сигнализирующих о такой ошибке, вместо этого выполняется программное прерывание с вектором 0.

Команда деления целых чисел со знаком idiv отличается от команды div только тем, что она учитывает знаки обоих операндов. В случае положительного результата команда аналогична команде div, за исключением того, что максимальное значение частного соответственно равно 127 и 32 767 для байтов и слов. Если результат отрицателен, частное усекается, а остаток имеет тот же знак, что и делимое. Минимальные значения частных для отрицательных результатов — −128 и −32 768 для байтов и слов.

При выполнении деления со знаком возникает проблема в случае, если делимое — байтовый операнд. Когда возникает необходимость разделить байтовое значение на байтовое, команда деления требует, чтобы делимое было 16-разрядным и занимало регистр AX. Эту проблему можно решить, применив команду преобразования байта в слово cbw. При этом из регистра AL берется число и расширяется его знак в регистр AH. Команда cbw загружает в регистр AX 16-битовое число, равное значению байта в регистре AL. Команда cwd выполняет аналогичную функцию для преобразования слова в двойное слово, расширяя знак слова из регистра AX в регистр DX. Эти две команды расширяют операнды до выполнения целого деления со знаком.

Для варианта целого деления без знака при тех же условиях знак не нужен, и его не надо расширять в старшую часть делимого. В этом случае можно просто обнулить регистр AH или DX перед делением.

Рассмотрим консольное приложение, выполняющее знаковое деление 64-битового операнда на 32-разрядный (листинг 2.17). Частное будет находиться в регистре EAX, а остаток — в регистре EDX.

Листинг 2.17. Программа, выполняющая деление 64-битового операнда на 32-разрядный

```
386
.model flat, stdcall
  option casemap :none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib
.data
  conTitle      DB  "Sign division", 0
  mes           DB  "The result of sign division 2788/(-360) = ", 0
  len_mes       EQU $-mes

  charBuf       DB  "    ", 0
  len_charBuf   DD  $-charBuf
  id1           DQ  2788
  id2           DD -360
  ires          DQ  0
  sign          DB  "+"
  lpFmt         DB  "%d", 0
  lpFmtMix      DB  "%c%d", 0

  readBuf       DB  ?
  lenReadBuf    DD  1
  ;
  hStdIn        DD  0
  hStdOut       DD  0
  ;
  chrsRead      DD  0
  chrsWritten   DD  0
  ;
```

```
  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11
.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex
  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex

  call    getout_hndl
  call    getinp_hndl

  ; операция деления 64-битовой переменной id1
  ; на 32-разрядный делитель id2

  mov     EAX, DWORD PTR id1
  mov     EDX, DWORD PTR id1+4
  mov     EBX, id2
  idiv    EBX

  ; сохранить частное по адресу ires+4
  ; и остаток по адресу ires

  mov     DWORD PTR ires+4, EAX
  mov     DWORD PTR ires, EDX

  ; преобразование значения частного в строку

  push    DWORD PTR ires+4
  push    offset lpFmt
  push    offset charBuf
  call    wsprintf
  add     ESP, 12

  push    EBX
```

```
mov     EBX, offset mes
mov     ECX, len_mes
call    write_con
pop     EBX

; вывод на экран значения частного

push    EBX
mov     EBX, offset charBuf
mov     ECX, len_charBuf
call    write_con
pop     EBX

; очистка строкового буфера

call    clear_buf

; преобразование в строку остатка от деления

push    DWORD PTR ires
push    DWORD PTR sign
push    offset lpFmtMix
push    offset charBuf
call    wsprintf
add     ESP, 16

; вывод на экран значения остатка от деления

push    EBX
mov     EBX, offset charBuf
mov     ECX, len_charBuf
call    write_con
pop     EBX

; ожидание ввода с консоли и выход

call    read_con
```

```
ex:
  push    0
  call    ExitProcess

; подпрограммы

getout_hndl proc
  push    STD_OUTP_HNDL
  call    GetStdHandle
  mov     hStdOut; EAX
  ret
getout_hndl endp

getinp_hndl proc
  push    STD_INP_HNDL
  call    GetStdHandle
  mov     hStdIn, EAX
  ret
getinp_hndl endp

write_con proc
  push    0
  push    chrsWritten
  push    ECX
  push    EBX
  push    hStdOut
  call    WriteConsoleA
  ret
write_con endp

read_con proc
  push    0
  push    chrsRead
  push    lenReadBuf
  push    offset readBuf
  push    hStdIn
  call    ReadConsoleA
  ret
read_con endp
```

```
clear_buf proc
  cld
  lea     EDI, BYTE PTR charBuf
  mov     ECX, len_charBuf
  mov     AL, 20h
  rep     stosb
  ret
clear_buf endp
end start
```

На рис. 2.6 изображено окно работающего приложения.

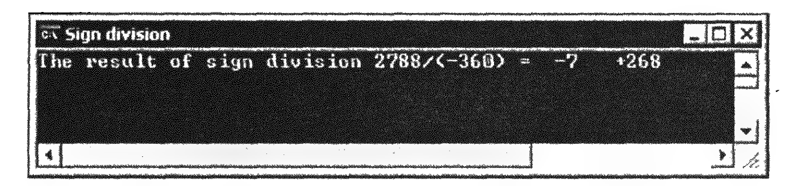


Рис. 2.6. Окно приложения, выполняющего деление двух чисел со знаком

Результатом деления целочисленной переменной, равной 2788, на −360 является число −7 с остатком 268.

Самые ранние модели процессоров фирмы Intel не имели аппаратной поддержки для операций с плавающей точкой. Все операции такого типа выполнялись как процедуры, в которые входили обычные арифметические команды. Для ранних моделей был разработан дополнительный кристалл, получивший название математического сопроцессора. В его состав входили команды, с помощью которых операции с плавающей точкой выполнялись намного быстрее, чем при использовании процедуры из обычных арифметических команд.

Начиная с процессоров Pentium, математический сопроцессор как отдельное устройство перестал существовать. Вместо него в состав процессоров входит блок операций с плавающей точкой (FPU — Floating-Point Unit), однако он программируется как отдельный модуль.

Сопроцессор добавляет арифметические возможности в систему, но не замещает ни одну команду основного процессора. Команды `add`, `sub`, `mul` и `div`, описанные ранее, выполняются процессором, а математический сопроцессор выполняет дополнительные, более эффективные команды арифметической обработки.

С точки зрения программиста, система с сопроцессором выглядит как единый процессор с большим набором команд.

Программная модель сопроцессора может быть представлена как совокупность регистров. Они могут быть разделены на три группы:

1. Регистры стека сопроцессора. Их 8, и они именуются как `ST(0)`, `ST(1)`, `ST(2)` ... `ST(7)`. Числа с плавающей точкой запоминаются как 80-битовые числа расширенного формата. Стек регистров организован по принципу LIFO (Last-In, First-Out — последним пришел, первым ушел). Регистр `ST(0)` всегда указывает на вершину стека. Вновь поступающие в сопроцессор числа добавляются в вершину стека.

2. Служебные регистры, в число которых входят регистр состояния, отражающий информацию о состоянии процессора, управляющий регистр (для управления режимами работы сопроцессора) и регистр состояния тегов, отражающий состояние регистров `ST(0)` ... `ST(7)`.

3. Регистр-указатель данных и регистр-указатель команд, предназначенные для обработки исключительных ситуаций.

К любому из вышеперечисленных регистров программа может получить доступ либо напрямую, либо косвенно. Для программирования сопроцессора в основном используются регистры `ST(0)` ... `ST(7)` и биты `C0`, `C1`, `C2` и `C3` регистра состояния.

Регистры сопроцессора функционируют как обычный стек основного процессора. Но у этого стека имеется ограниченное число позиций — только 8. Сопроцессор имеет еще один регистр, труднодоступный для программиста. Он представляет собой слово, содержащее "метки" каждой позиции стека. Такой регистр позволяет сопроцессору отслеживать, какие из позиций стека используются, а какие свободны. Любая попытка поместить объект в стек на уже занятую позицию приведет к возникновению исключительной ситуации — недействительной операции.

Программа заносит данные в стек сопроцессора с помощью команды загрузки, которая помещает данные в вершину стека. Если число в памяти записано не во временном действительном формате, то сопроцессор преобразует его в 80-битовое представление во время выполнения команды загрузки.

Команды записи извлекают значение из стека сопроцессора и помещают их в память. Если необходимо преобразование формата данных, сопроцессор выполняет его как часть операции записи. Некоторые формы операции записи оставляют вершину стека нетронутой для дальнейших действий.

После того как данные помещены в стек сопроцессора, они могут быть использованы любой командой. Инструкции процессора допускают как действия между регистрами, так и действия между памятью и регистрами. По аналогии с основным процессором, из любых двух операндов арифметической операции один должен находиться в регистре. У сопроцессора один из операндов должен быть всегда верхним элементом стека, а другой операнд может быть взят из памяти, либо из стека регистров.

Стек регистров всегда должен быть приемником результата любой арифметической операции. Непосредственно записать результат в память той же командой, которая выполнила вычисления, процессор числовой обработки не может. Для пересылки операнда обратно в память необходимо воспользоваться отдельной командой записи или командой извлечения из стека с последующей записью в память.

Все команды математического сопроцессора начинаются с буквы F, чтобы отличить их от команд основного процессора. Условно команды сопроцессора можно разделить на несколько групп:

- команды записи и чтения;
- команды сложения/вычитания;
- команды умножения/деления;
- команды сравнения;
- команды трансцендентных функций;
- дополнительные команды.

Рассмотрим более подробно каждую из этих групп команд.

Команда записи имеет два варианта. Одна из модификаций этой команды извлекает число из вершины стека и записывает его в ячейку памяти. Выполняя эту команду, сопроцессор преобразует данные из временного действительного формата в желаемую внешнюю форму. Для этой команды определены коды операций `fst` и `fist`. Эти команды позволяют занести значение вершины стека в любой регистр внутри стека.

Второй вариант команды записи кроме записи данных изменяет положение указателя стека. Команды `fstp` (как и команды `fistp` и `fbstp`), выполняя ту же операцию записи данных из сопроцессора в память, извлекают число из стека. Эта модификация команд поддерживает все внешние типы данных.

Команда замены `fxch` — следующая команда в группе команд пересылки данных. Она меняет местами содержимое вершины стека с содержимым другого регистра стека. В качестве операнда этой команды может использоваться только другой элемент стека. Обменять содержимое вершины стека и ячейки памяти эта команда не позволяет. Для этого требуется выполнить несколько команд. Сопроцессор может в одной команде выполнять чтение из памяти или запись в память, но не то и другое одновременно.

Команды чтения или загрузки помещают данные в вершину стека сопроцессора. Основной из них является команда `fld`. Для загрузки целых чисел используется модификация команды — `fild`.

Следующей группой, которую мы рассмотрим, являются команды сложения/вычитания. Каждая из этих команд находит сумму или разность реги-

стра ST(0) и другого операнда. Результат операции всегда помещается в регистр сопроцессора. Далее представлена мнемоника этих команд.

```
fadd    ST(0), ST(1)
fadd    ST(0), ST(2)
fadd    ST(2), ST(0)
fiadd   WORD_INTEGER
fiadd   SHORT_INTEGER
fadd    SHORT_REAL
fadd    LONG_REAL
faddp   ST(2), ST(0)
fsub    ST(0), ST(2)
fisub   WORD_INTEGER
fsubp   ST(2), ST(0)
fsubr   ST(2), ST(0)
fisubr  SHORT_INTEGER
fsubrp  ST(2), ST(0)
```

Операндами этих команд могут быть либо два регистра стека сопроцессора, либо регистр стека и ячейка памяти. В качестве исходных данных ячейки памяти используются слово и короткое слово разрядностью 16 и 32 бита.

Фрагмент программного кода (листинг 2.18) демонстрирует операцию сложения двух целых чисел.

Листинг 2.18. Фрагмент программного кода, выполняющий сложение двух целых чисел

```
  ...
.data
  IX1     DD  43
  IX2     DD -34
  ISUM    DD  0
.code
   ...
  finit
  fild    DWORD PTR IX1
  fiadd   DWORD PTR IX2
  fistp   DWORD PTR ISUM
  fwait
   ...
```

Первая команда finit инициализирует сопроцессор. Команда fild загружает значение целочисленной переменной IX1 в вершину стека. Команда fiadd выполняет операцию сложения содержимого стека и ячейки памяти IX2, оставляя результат в вершине стека. После этого команда fistp помещает результат в ячейку памяти ISUM, одновременно удаляя это значение из вершины стека.

В листинге 2.19 представлен фрагмент кода для суммирования вещественных чисел.

Листинг 2.19. Фрагмент кода, выполняющий суммирование вещественных чисел

```
  ...
.data
  X1      DD  43.7
  X2      DD -34.11
  FSUM    DD  0
.code
   ...
  finit
  fld     DWORD PTR X1
  fadd    DWORD PTR X2
  fstp    DWORD PTR FSUM
  fwait
   ...
```

Первая команда fld загружает вещественное число X1 из памяти в вершину стека сопроцессора. Команда fadd вычисляет сумму значений вершины стека ST(0) и ячейки памяти, содержащей значение X2. Результат операции сохраняется в вершине стека сопроцессора. Наконец, команда fstp сохраняет значение суммы в переменной FSUM. Вершина стека ST(0) очищается.

В следующем примере мы рассмотрим применение команд загрузки, сложения и сохранения для нахождения суммы элементов целочисленного массива из семи элементов. Подобные задачи очень часто приходится решать на практике. Исходный текст программы приведен в листинге 2.20.

Листинг 2.20. Программа нахождения суммы элементов целочисленного массива

```
.386
.model flat, stdcall
  option  casemap :none
```

```
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib
.data

  conTitle      DB  "Sum of integers in array", 0
  mes1          DB  "Array: ", 0
  len_mes1      EQU $-mes1

  mes2          DB  0dh, 0ah, "Sum of elements = ", 0
  len_mes2      EQU $-mes2

  charBuf       DB  "       ", 0
  len_charBuf   DD  $-charBuf
  iarray        DD -9, 3, -6, 2, 11, -5
  larray        EQU ($-iarray)/4
  ISUM          DD  0
  lpFmt         DB  "%d", 0

  readBuf       DB  ?
  lenReadBuf    DD  1

  hStdIn        DD  0
  hStdOut       DD  0

  chrsRead      DD  0
  chrsWritten   DD  0

  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11

.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex
```

```
  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex

  call    getout_hndl
  call    getinp_hndl

  push    EBX
  mov     EBX, offset mes1
  mov     ECX, len_mes1
  call    write_con
  pop     EBX

  mov     ESI, offset iarray
  mov     ECX, larray
show_next:
  push    ESI
  push    ECX
  push    DWORD PTR [ESI]
  push    offset lpFmt
  push    offset charBuf
  call    wsprintf
  add     ESP, 12

  push    EBX
  mov     EBX, offset charBuf
  mov     ECX, len_charBuf
  call    write_con
  pop     EBX

  call    clear_buf
  pop     ECX
  pop     ESI
  add     ESI, 4
  loop    show_next

; нахождение суммы элементов массива
```

```
  mov     ECX, larray
  mov     ESI, offset iarray
  finit
  fild    DWORD PTR [ESI]
next:
  fiadd   DWORD PTR [ESI+4]
  add     ESI, 4
  loop    next
  fistp   DWORD PTR ISUM
  fwait

  ; преобразование суммы в строку символов

  push    DWORD PTR ISUM
  push    offset lpFmt
  push    offset charBuf
  call    wsprintf
  add     ESP, 12

  ; вывод заголовка

  push    EBX
  mov     EBX, offset mes2
  mov     ECX, len_mes2
  call    write_con
  pop     EBX

  ; вывод значения суммы на экран

  push    EBX
  mov     EBX, offset charBuf
  mov     ECX, len_charBuf
  call    write_con
  pop     EBX
  call    clear_buf

  ; ожидание ввода с консоли и выход из программы

  call    read_con
```

```
ex:
  push    0
  call    ExitProcess
   ...
;----------- Процедуры -----------
   ...
```

Для нахождения суммы элементов массива воспользуемся следующим простым алгоритмом: сперва загрузим в вершину стека первый элемент массива, после чего будем прибавлять к нему в каждой итерации значение последующего элемента. Количество итераций, равное размеру массива, помещаем в регистр ECX. После того, как сумма найдена, помещаем ее в переменную ISUM командой fistp.

Вид окна работающего приложения представлен на рис. 2.7.

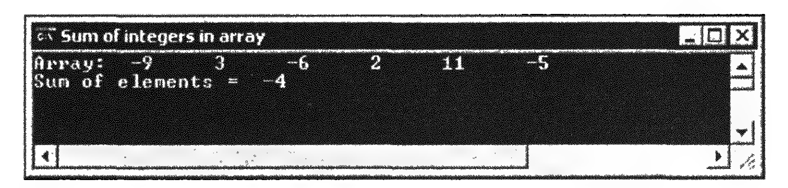


Рис. 2.7. Окно приложения, вычисляющего сумму элементов массива целых чисел с использованием команд математического сопроцессора

Следующая группа команд, которую мы рассмотрим, — команды умножения/деления целых и вещественных чисел. Примеры их вызовов можно представить в виде фрагмента кода:

```
WORD_INTEGER    LABEL WORD
SHORT_INTEGER   LABEL DWORD
SHORT_REAL      LABEL DWORD
LONG_REAL       LABEL QWORD

fmul     SHORT_REAL
fimul    WORD_INTEGER
fmulp    ST(2), ST(0)
fdiv     ST(0), ST(2)
fidiv    SHORT_INTEGER
fdivp    ST(2), ST(0)
fdivr    ST(0), ST(2)
fidivr   WORD_INTEGER
fdivrp   ST(2), ST(0)
```

Как и в случае операций сложения и вычитания, в качестве операндов этих команд используются либо два регистра сопроцессора, либо регистр стека и ячейка памяти. Показать использование команд сложения/умножения лучше всего на примере. Программный код этого примера более сложный и демонстрирует способы применения различных команд сопроцессора. Необходимо найти числовое значение величины Z вещественного типа, определяемой формулой (X - Y)/(X + Y). Фрагмент кода программы приведен в листинге 2.21.

Листинг 2.21. Пример кода, использующий различные команды сопроцессора

```
  ...
.data
  X       DD  1.3
  Y       DD -7.8
  Z       DD  0.0
.code
   ...
 finit
  fld     DWORD PTR X
  fadd    DWORD PTR Y
  fld     DWORD PTR X
  fsub    DWORD PTR Y
  fxch    st(1)
  fdiv    st(1), st(0)
  fxch    st(1)
  fstp    DWORD PTR Z
  fwait
   ...
```

Проведем детальный анализ примера. Вычисление выражения (X - Y)/(X + Y) выполним в три этапа. Первый шаг — вычисление знаменателя при помощи команд:

```
fld     DWORD PTR X
fadd    DWORD PTR Y
```

В вершину стека (регистр ST(0)) загружается значение переменной X. Далее к этому значению прибавляется значение переменной Y. В результате выполнения этих двух команд в вершине стека будет находиться сумма X и Y.

Следующие две команды выполняют вычисление разности X и Y. Для этого в вершину стека загружается значение X, затем производится вычитание значения Y:

```
fld     DWORD PTR X
fsub    DWORD PTR Y
```

Далее надо быть очень внимательным. После выполнения первых пяти команд нашего фрагмента в регистре ST(0) находится разность X и Y. Так как стек сопроцессора организован в виде циклического буфера, то ранее вычисленное значение X + Y переместилось в регистр стека ST(1). Чтобы разделить разность переменных X и Y на их сумму, поменяем значения в регистрах ST(0) и ST(1) и выполним операцию деления содержимого регистра ST(1) на значение в регистре ST(0):

```
fxch    st(1)
fdiv    st(1), st(0)
```

После выполнения этих команд в регистре ST(1) находится вычисленное значение величины Z. Чтобы записать значение ST(1) в переменную Z, выполним следующие команды:

```
fxch    st(1)
fstp    DWORD PTR Z
```

Как и в наборе команд процессора, у математического сопроцессора имеются команды, выполняющие сравнение двух чисел. Далее приводится мнемоника команд сравнения:

```
WORD_INT      LABEL  WORD
SHORT_INT     LABEL  DWORD
SHORT_REAL    LABEL  DWORD
LONG_REAL     LABEL  QWORD

fcom
fcomp   ST(2)
ficom   WORD_INT
fcom    SHORT_REAL
fcomp
```

```
ficomp  SHORT_INT
fcomp   LONG_REAL
fcompp
ftst
fxam
```

Сопроцессор устанавливает в соответствии с результатом сравнения флаги состояния. Перед тем как опросить флаги состояния, программа должна считать *слово состояния* в память. Самый простой способ — загрузить флаги состояния в регистр AH, а затем, чтобы облегчить себе задачу проверки условия, — в регистр флагов процессора.

В операции всегда участвует вершина стека, поэтому в команде надо указать только один регистр или ячейку памяти. После сравнения в слове состояния процессора содержится результат этой операции. При этом бит C0 помещается на место флага переноса CF, C2 — на место бита четности PF, а C3 — на место ZF.

Для отражения результата сравнения необходимы только два бита состояния: C3 и C0. В табл. 2.1 приводится соотношение сравниваемых операндов и битов состояния.

***Таблица 2.1.** Соотношение сравниваемых операндов и битов состояния*

C3	C0	Результат
0	0	ST > источник
0	1	ST < источник
1	0	ST = источник
1	1	ST и источник несравнимы

Следующая программа (листинг 2.22) сравнивает два вещественных числа и выводит результат сравнения на экран.

Листинг 2.22. Программа, выполняющая сравнение двух вещественных чисел

```
.386
.model flat, stdcall
  option  casemap :none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
```

```
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib

.data
  conTitle     DB "Comparison of two reals", 0
  mes     DB "The result of comparison X = -87.34 and Y = -136.57 : ",0
  len_mes      EQU $-mes
  xgey         DB  "X >= Y", 0
  len_xgey     EQU $-xgey
  xly          DB  "X < Y", 0
  len_xly      EQU $-xly

  X             DD -87.34
  Y             DD -136.57
  Flag          DD  0

  readBuf       DB  ?
  lenReadBuf    DD  1
  hStdIn        DD  0
  hStdOut       DD  0
  chrsRead      DD  0
  chrsWritten   DD  0
  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11

.code
start:
  call    AllocConsole
  test    AX, EAX
  jz      ex
  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex
  call    getout_hndl
  call    getinp_hndl
```

```
  push    EBX
  mov     EBX, offset mes
  mov     ECX, len_mes
  call    write_con
  pop     EBX

  finit
  fld     DWORD PTR X
  fcomp   DWORD PTR Y
  fstsw   AX
  sahf
  jna     x_less_y
  mov     DWORD PTR Flag, 1
x_less_y:
  cmp     Flag, 0
  je      WriteXLY

  push    EBX
  mov     EBX, offset xgey
  mov     ECX, len_xgey
  call    write_con
  pop     EBX
  jmp     rcon
WriteXLY:
  push    EBX
  mov     EBX, offset xly
  mov     ECX, len_xly
  call    write_con
  pop     EBX

; ожидание ввода с консоли и выход

rcon:
  call    read_con
ex:
  push    0
  call    ExitProcess

;------ Процедуры ----
```

```
getout_hndl proc
  push    STD_OUTP_HNDL
  call    GetStdHandle
  mov     hStdOut, EAX
  ret
getout_hndl endp

getinp_hndl proc
  push    STD_INP_HNDL
  call    GetStdHandle
  mov     hStdIn, EAX
  ret
getinp_hndl endp

write_con proc
  push    0
  push    chrsWritten
  push    ECX
  push    EBX
  push    hStdOut
  call    WriteConsoleA
  ret
write_con endp

read_con proc
  push    0
  push    chrsRead
  push    lenReadBuf
  push    offset readBuf
  push    hStdIn
  call    ReadConsoleA
  ret
read_con endp
end start
```

Операция сравнения в программе реализована в следующих строках программного кода:

```
finit
fld     DWORD PTR X
```

```
fcomp   DWORD PTR Y
fstsw   AX
sahf
jna     x_less_y
```

После инициализации сопроцессора командой `finit` в вершину стека помещается переменная `X`. Команда `fcomp` сравнивает число в вершине стека с переменной в памяти и в зависимости от результата устанавливает биты в слове состояния сопроцессора.

На рис. 2.8 изображено окно работающего приложения.

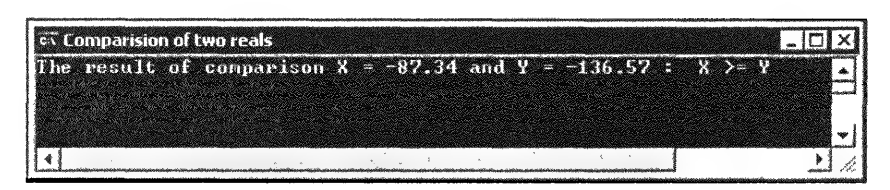


Рис. 2.8. Окно приложения, сравнивающего два вещественных числа

Кроме команды `fcomp`, существуют и другие варианты команды сравнения `fcom`, в частности — версия `ficomp` для сравнения целых чисел. Далее представлен фрагмент программы для сравнения двух целых чисел (листинг 2.23).

Листинг 2.23. Фрагмент программы сравнения двух целых чисел

```
 ...
.data
 conTitle    DB "Comparison of two ints", 0
 mes         DB "The result of comparison IX = -65 and IY = -13 : ", 0
 len_mes        EQU $-mes
 xgy            DB  "X > Y", 0
 len_xgy        EQU $-xgy
 xly            DB  "X < Y", 0
 len_xly        EQU $-xly
 xey            DB  "X = Y", 0
 len_xey        EQU $-xey
 IX             DD -65
 IY             DD  13

 STATUS         LABEL WORD
```

```
  STATUS_WORD    DW  ?

  readBuf        DB  ?
  lenReadBuf     DD  1
  hStdIn         DD  0
  hStdOut        DD  0
  chrsRead       DD  0
  chrsWritten    DD  0
  STD_INP_HNDL   DD -10
  STD_OUTP_HNDL  DD -11
.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex

  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex

  call    getout_hndl
  call    getinp_hndl

  push    EBX
  mov     EBX, offset mes
  mov     ECX, len_mes
  call    write_con
  pop     EBX

  finit
  fild    DWORD PTR IX
  ficomp  DWORD PTR IY
  fstsw   STATUS_WORD
  mov     AH, BYTE PTR STATUS+1
  sahf
  jb      x_less_y
  jne     x_great_y
  jmp     WriteXEY
```

```
x_less_y:
  jmp     WriteXLY
x_great_y:
  jmp     WriteXGY

  ; вывод сообщения "X = Y"

WriteXEY:
  push    EBX
  mov     EBX, offset xey
  mov     ECX, len_xey
  call    write_con
  pop     EBX
  jmp     rcon

  ; вывод сообщения "X < Y"

WriteXLY:
  push    EBX
  mov     EBX, offset xly
  mov     ECX, len_xly
  call    write_con
  pop     EBX
  jmp     rcon

  ; вывод сообщения "X > Y"

WriteXGY:
  push    EBX
  mov     EBX, offset xgy
  mov     ECX, len_xgy
  call    write_con
  pop     EBX

  ; ожидание ввода с консоли и выход

rcon:
  call    read_con
```

```
ex:
  push    0
  call    ExitProcess
   ...
```

В этом фрагменте особенно интересны следующие строки:

```
finit
fild    DWORD PTR IX
ficomp  DWORD PTR IY
fstsw   STATUS_WORD
mov     AH, BYTE PTR STATUS+1
sahf
jb      x_less_y
jne     x_great_y
jmp     WriteXEY
```

После инициализации сопроцессора с помощью команды `fild` в его стек загружается первое число `IX`. После сравнения чисел `IX` и `IY` результат сравнения записывается в переменную `STATUS_WORD` командой `fstsw`. После перемещения в регистр `AH` старшего байта `STATUS_WORD` можно проанализировать флаги `CF(C0)`, `PF(C2)`, `ZF(C3)`. Далее, следуя соотношениям в табл. 2.1, используем команды условных переходов для вывода в окно приложения соответствующих сообщений.

Рассмотрим еще один пример программного кода. Необходимо подсчитать, сколько раз встречается в массиве целых чисел определенное число. Исходный текст программного кода приведен в листинге 2.24.

Листинг 2.24. Фрагмент программы подсчета количества вхождений целого числа в массив

```
 ...
.data
 conTitle     DB  "Counting of separate int in array", 0
 mes1         DB  "Array: ", 0
 len_mes1     EQU $-mes1
 mes2         DB  0dh, 0ah, "Number = ", 0
 len_mes2     EQU $-mes2
 mes3         DB  0dh, 0ah, "Found times = ", 0
 len_mes3     EQU $-mes3
```

```
  charBuf        DB  "      ", 0
  len_charBuf    DD  $-charBuf
  iarray         DD -9, 3, -5, 2, 11, 7, -5, 78, -5, 12
  larray         EQU ($-iarray)/4
  lpFmt          DB  "%d", 0
  cnt            DD  0
  num            DD -5
  readBuf        DB  ?
  lenReadBuf     DD  1
  hStdIn         DD  0
  hStdOut        DD  0
  chrsRead       DD  0
  chrsWritten    DD  0
  STD_INP_HNDL   DD -10
  STD_OUTP_HNDL  DD -11
.code
start:
  call     AllocConsole
  test     EAX, EAX
  jz       ex
  push     offset conTitle
  call     SetConsoleTitleA
  test     EAX, EAX
  jz       ex

  call     getout_hndl
  call     getinp_hndl

  push     EBX
  mov      EBX, offset mes1
  mov      ECX, len_mes1
  call     write_con
  pop      EBX

  mov      ESI, offset iarray
  mov      ECX, larray
show_next:
  push     ESI
```

```
push    ECX
push    DWORD PTR [ESI]
push    offset lpFmt
push    offset charBuf
call    wsprintf
add     ESP, 12

push    EBX
mov     EBX, offset charBuf
mov     ECX, len_charBuf
call    write_con
pop     EBX
call    clear_buf
pop     ECX
pop     ESI
add     ESI, 4
loop    show_next

push    EBX
mov     EBX, offset mes2
mov     ECX, len_mes2
call    write_con
pop     EBX

push    DWORD PTR num
push    offset lpFmt
push    offset charBuf
call    wsprintf
add     ESP, 12

push    EBX
mov     EBX, offset charBuf
mov     ECX, len_charBuf
call    write_con
pop     EBX

; подсчитать, сколько раз встречается элемент в массиве

mov     DWORD PTR cnt, 0
```

```
  lea     ESI, iarray
  mov     ECX, larray
  finit

  ; загрузка исходного числа в вершину стека сопроцессора

  fild    DWORD PTR num
next_cmp:
  ficom   DWORD PTR [ESI]
  fstsw   AX
  sahf
  jne     skip

  ; если значение в вершине стека равно элементу массива,
  ; увеличить содержимое счетчика

  inc     cnt
skip:
  add     ESI, 4
  loop    next_cmp

  ; преобразовать результат подсчета в строку

  push    DWORD PTR cnt
  push    offset lpFmt
  push    offset charBuf
  call    wsprintf
  add     ESP, 12

  push    EBX
  mov     EBX, offset mes3
  mov     ECX, len_mes3
  call    write_con
  pop     EBX
  push    EBX
  mov     EBX, offset charBuf
  mov     ECX, len_charBuf
  call    write_con
  pop     EBX
  call    clear_buf
```

```
;----- Процедуры -----
   ...
```

Вид окна работающего приложения дан на рис. 2.9.

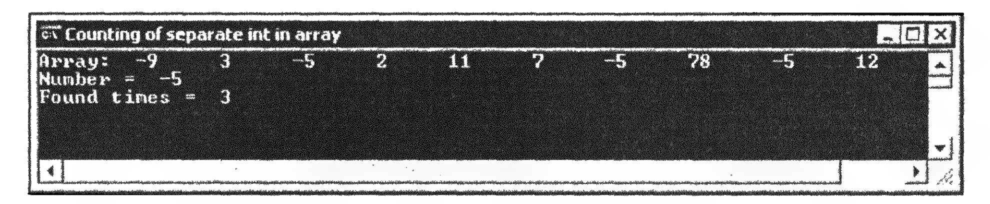


Рис. 2.9. Окно приложения, выполняющего подсчет количества вхождений целого числа в массив

Команды сравнения с извлечением из стека обеспечивают удобный способ его очистки. Математический сопроцессор не имеет команды, которая бы легко извлекала операнд из стека, вместо нее можно использовать команды сравнения с извлечением из стека. Эти команды также изменяют и регистр состояния, поэтому их нельзя применять, если биты состояния необходимо использовать в дальнейшей работе. Но в большинстве случаев эти команды позволяют быстро извлечь из стека один или два операнда.

Существуют две специальные команды сравнения. Это команда сравнения содержимого вершины стека с нулем `ftst`, с помощью которой можно быстро определить знак содержимого вершины стека, и команда `fxam`.

Команда `fxam` анализирует операнд в вершине стека сопроцессора `ST(0)` и устанавливает определенным образом биты состояния `C0-C3` в регистре состояния. Иными словами, команда сравнивает операнд в стеке со стандартными типами числовых значений, определенных для математического сопроцессора. К стандартным типам фирма Intel относит корректные положительные и отрицательные вещественные числа, ноль, нечисла (Not-A-Number — NAN), числа неизвестного формата, бесконечность (Infinity), положительные и отрицательные денормализованные числа (Denormal), пустое значение в регистре `ST(0)` (Empty).

Приведем пример программы, выполняющей анализ числа при помощи команды `fxam`. Это консольное приложение, написанное на Delphi. Программа ожидает ввода с клавиатуры целого числа и анализирует его знак и равенство нулю. Результат сравнения выводится на экран дисплея. Пусть вас не смущает то, что для иллюстрации процесса вычислений мы используем встроенный ассемблер Delphi. По способу передачи параметров и возврата результата наша процедура эквивалентна внешней процедуре на языке TASM фирмы Borland. Исходный текст приложения приведен в листинге 2.25.

Листинг 2.25. Программа, выполняющая анализ введенных с клавиатуры целых чисел

```
program fxamex;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  i1, ires: Integer;
  c1: Char;

function CheckFxam(pi: PInteger): Integer; assembler;
asm
  finit
  fild   DWORD PTR [pi]
  fxam
  xor     EAX, EAX
  fstsw   AX
  and     AX, 4700h
  shr     EAX, 6
  sal     AH, 5
  or      AL, AH
  shr     AL, 2
  xor     AH, AH
  fwait
end;

begin
  repeat
    WriteLn;
    Write('Enter integer value: ');
    ReadLn(i1);
    ires := CheckFxam(@i1);
    case ires of
      4:  WriteLn('-> Positive Normal');
      6:  WriteLn('-> Negative Normal');
```

```
      8:  WriteLn('-> Zero');
      10: WriteLn('-> Zero');
    else
      WriteLn('-> Other');
    end;
    Write('Press c to continue or other key to exit...');
    ReadLn(c1);
  until c1 <> 'c';
end.
```

Приведенный пример является довольно сложным и требует дополнительного анализа. Основные вычисления выполняются в процедуре CheckFxam. В качестве входного параметра процедура принимает адрес переменной целочисленного типа. Анализ переменной выполняется двумя командами:

```
fild    DWORD PTR [pi]
fxam
```

После выполнения этих команд биты C3-C0 устанавливаются определенным образом в регистре состояния сопроцессора. Для удобства работы сохраним значения этих битов в предварительно обнуленном регистре AX:

```
xor     EAX, EAX
fstsw   AX
```

Сгруппируем C3-C0 в регистре AX таким образом, чтобы они заняли позиции младших битов в регистре AL. Это выполняется с помощью команд:

```
and     AX, 4700h
shr     EAX, 6
sal     AH, 5
or      AL, AH
shr     AL, 2
```

Последнее, что нужно сделать — вернуть результат в регистре EAX в основную программу, не забыв перед этим обнулить регистр AH. Возвращаемое значение находится в диапазоне от 0 до 15. Основная программа анализирует знак целого числа и его равенство нулю. Это соответствует значениям 4, 6, 8 и 10 в регистре EAX. Остальные значения для упрощения анализа программы не рассматриваются.

Окно работающего приложения показано на рис. 2.10.

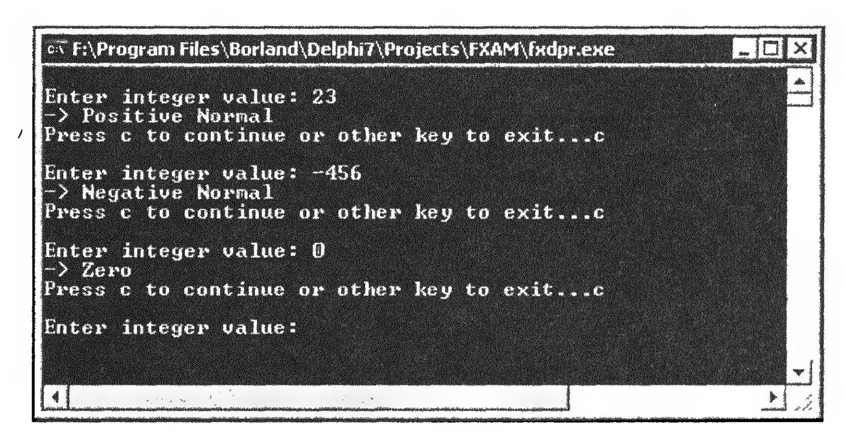


Рис. 2.10. Окно приложения, выполняющего анализ введенных с клавиатуры чисел

Если при арифметической обработке не делать чего-либо из ряда вон выходящего и не работать на пределе разрядной сетки сопроцессора, то вряд ли вам будет интересна команда `fxam`. Мы не будем детально рассматривать реакцию сопроцессора на исключительные ситуации, иногда возникающие при вычислениях. Это очень большая тема, и желающие могут обратиться к фирменному руководству Intel по 387 процессору.

Следующей группой команд, на которые мы обратим наше внимание, являются команды, вычисляющие значения степенных и тригонометрических функций. Эти команды позволяют сопроцессору вычислять сложные математические выражения, такие как логарифмы, экспоненты и тригонометрические функции. Далее приведен список этих команд:

```
fsqrt
fscale
fprem

fxtract
fabs
fchs
fsin
fcos
fsincos
fptan
fpatan
```

```
f2xm1
fyl2x
fyl2xp1
```

Наличие команд трансцендентных функций значительно усиливает вычислительную мощность процессора. Результат вычислений таких функций имеет высокую точность. Необходимо учитывать тот факт, что аргументы тригонометрических функций задаются в радианах. Если, к примеру, мы хотим вычислить синус угла А, заданного в градусах, то для перевода в радианы можно использовать формулу

A_РАД = A * PI / 180,

где А_РАД — величина угла в радианах, А — величина угла в градусах.

Рассмотрим небольшой пример, в котором будут вычисляться синус и косинус угла. В качестве каркаса приложения будем использовать консольное приложение, сгенерированное Мастером приложений Delphi 7. Операторы ввода-вывода такого приложения позволяют легко манипулировать любыми типами данных, а это упрощает исходный текст программы. В этом приложении будем использовать две простейшие команды Delphi — `ReadLn` и `WriteLn`. Первая из этих команд читает ввод с клавиатуры, а вторая выводит значения на экран. Программа, исходный текст которой приведен в листинге 2.26, довольно проста.

Листинг 2.26. Программа, вычисляющая синус и косинус угла

```
program sincos;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  angle: Single;
  angleRad: Single;
  Sinus, Cosinus: Single;

procedure SinCos(angle: Single); assembler;
asm
  finit
  fld     DWORD PTR angleRad
  fld     DWORD PTR angleRad
```

```
  fsin
  fstp    DWORD PTR Sinus
  fcos
  fstp    DWORD PTR Cosinus
  fwait
end;

begin
  Write('Enter degrees: ');
  ReadLn(angle);
  AngleRad := angle*3.14/180;
  SinCos(angleRad);

  WriteLn('The angle in degrees = ', angle:5:2);
  WriteLn('Sinus of angle = ', Sinus:5:3);
  WriteLn('Cosinus of angle = ', Cosinus:5:3);
  ReadLn;
end.
```

В программе для вычисления значений синуса и косинуса используется процедура `SinCos` на языке ассемблера. Основная программа читает введенные посредством клавиатуры значения угла и, преобразовав полученное значение угла в радианы, вызывает процедуру вычисления синуса и косинуса `SinCos`. Команды `fsin` и `fcos` вычисляют значения синуса и косинуса угла, находящегося в вершине стека `ST(0)`. Команды не имеют операндов и возвращают результат в том же регистре `ST(0)`. Предыдущее значение регистра (значение угла) теряется после операции вычисления синуса. Вот почему возникает необходимость выполнения двух команд `fld` в процедуре!

На рис. 2.11 изображено окно приложения.

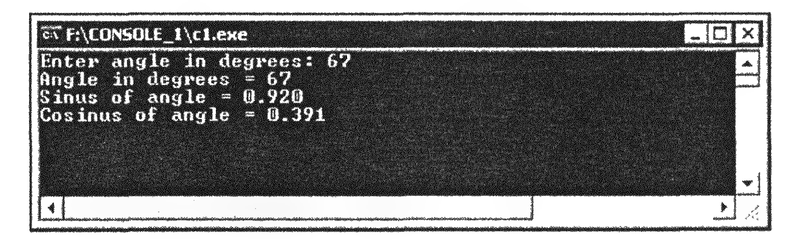


Рис. 2.11. Окно приложения, выполняющего вычисление синуса и косинуса угла

Среди команд вычисления значений тригонометрических функций есть `fsincos`. Эта команда вычисляет синус и косинус угла, находящегося в вершине стека сопроцессора `ST(0)`. Команда не использует операнды и возвращает результат в регистрах `ST(0)` и `ST(1)`. При этом в `ST(0)` помещается значение синуса, а в регистр `ST(1)` — косинуса. Изменим предыдущий пример так, чтобы можно было использовать команду `fsincos`.

Исходный текст программы представлен в листинге 2.27.

Листинг 2.27. Программа, вычисляющая синус и косинус угла с помощью функции FSINCOS

```
program c1;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  angle: Integer;
  angleRad: Single;
  Sinus, Cosinus: Single;

procedure SinCos(angle: Single); assembler;
asm
  finit
  fld     DWORD PTR angleRad
  fsincos
  fxch    st(1)
  fstp    DWORD PTR Sinus
  fstp    DWORD PTR Cosinus
  fwait
end;

begin
  Write('Enter angle in degrees: ');
  ReadLn(angle);
  AngleRad := angle*3.14/180;
  SinCos(angleRad);
  WriteLn('Angle in degrees = ', angle);
```

```
  WriteLn('Sinus of angle = ', Sinus:5:3);
  WriteLn('Cosinus of angle = ', Cosinus:5:3);
  ReadLn;
end.
```

Мы рассмотрели далеко не полный набор математических функций ассемблера. Главная цель — убедить читателя в полной мере использовать математический сопроцессор в программах на ассемблере. При желании можно модифицировать существующие или написать свои собственные математические алгоритмы, которых нет в языках высокого уровня.

2.3. Обработка строк и массивов данных

Преимущества ассемблера особенно сильно проявляются при обработке строк и массивов данных. Для выполнения таких операций была разработана целая группа команд, в терминологии Intel именуемая командами строковых примитивов. Под обработкой строк мы будем понимать выполнение следующих операций:

- сравнение двух строк;
- копирование строки-отправителя в строку-получатель;
- считывание строк из устройства или файла;
- запись строки в устройство или файл;
- определение размера строки;
- нахождение подстроки в заданной строке;
- объединение двух строк (*конкатенация*).

Операции над строками широко используются в языках высокого уровня. Ассемблерная реализация таких операций позволяет существенно повысить быстродействие программ на языках высокого уровня, особенно если требуется обработать большое число строк и массивов. Рассмотрим вначале основные команды языка ассемблера для обработки строк.

Программа представляет каждую строку массивом символов в памяти и может выполнять строковые операции над байтами, словами и двойными словами.

Строковые команды не применяют способы адресации, используемые другими командами. Они используют регистры `ESI` и `EDI`. В эти регистры помещается адрес первого элемента, с которого начинается обработка строки. Все строковые команды корректируют адрес после выполнения операции. Строка может состоять из нескольких символов, но в каждый момент времени команды обработки строк могут работать только с одним символом.

4 Зак. 1064

Автоматический *инкремент* (увеличение) или *декремент* (уменьшение) адреса операнда позволяет быстро обрабатывать строковые данные. ***Флаг направления*** (Direct Flag) в регистре состояния определяет направление обработки строк. Если он равен 1, то адрес уменьшается, а если он сброшен в 0, то адрес увеличивается. Сама величина инкремента или декремента адреса определяется размером операнда. Например, для символьных строк, в которых размер операндов равен 1 байту, команды обработки строк изменяют адрес на 1 после каждой операции. Если обрабатывается массив целых чисел, в котором каждый операнд занимает 4 байта, то строковые команды изменяют адрес на 4. После выполнения команды указатель адреса в регистре ESI или EDI ссылается на следующий элемент строки.

Рассмотрим представление строк в разных языках программирования. Наиболее часто используются *строки с завершающим нулем* (null-terminated strings). Они используются в языке C и в операционных системах Windows. Вот как определяется такая строка на языке ассемблера:

```
String_0     DB "NULL-TERMINATED STRING", 0
```

В языке Pascal (соответственно, в Delphi) в начале строки указывается ее размер. Элементы, расположенные за последним символом строки, считаются неопределенными. На языке ассемблера запись такой строки могла бы выглядеть так:

```
String_PAS   DB 0Dh, "STRING PASCAL"
```

Мы будет рассматривать в основном строки с завершающим нулем. Можно выделить пять основных команд для работы со строками. К этим командам относятся:

- movs — команда для перемещения строки данных из одного участка памяти в другой;
- lods — команда загрузки строки, адрес которой указан в регистре ESI, в регистр-аккумулятор EAX (AX, AL);
- stos — команда сохранения содержимого регистра EAX (AX, AL) в памяти по адресу, указанному в регистре EDI;
- cmps — команда сравнения строк, расположенных по адресам, содержащимся в регистрах ESI и EDI;
- scas — команда сканирования строк. Сравнивает содержимое регистра EAX (AX, AL) с содержимым памяти, определяемым регистром EDI.

Каждая команда обработки строк имеет три допустимых формата. Например, команда movs может иметь одно из представлений: movsb, movsw, movsd. Команда movsb может использоваться только для работы с однобайтовыми операндами, movsw — для работы со словами, а movsd — для работы с двой-

ными словами. Суффиксы b, w и d определяют шаг инкремента и декремента для индексных регистров ESI и EDI. Если команда используется в общем формате, то размерность операндов должна быть определена явно.

Перед выполнением команд строковых примитивов необходимо, чтобы в регистры ESI и/или EDI были загружены адреса обрабатываемых ячеек памяти.

Для выполнения повторяющихся операций со строками практически всегда используется *префикс повторения* rep. Это позволяет выполнить строковую операцию количество раз, определенное содержимым регистра ECX.

Может показаться удобным использовать комбинацию команд lods и stos для перемещения данных из одного места в другое, но для этой цели существует команда пересылки строки movs. Она считывает данные по адресу памяти, находящемуся в регистре ESI, и помещает их по адресу, указываемому регистром EDI. При этом содержимое регистров ESI и EDI изменяется так, чтобы указывать на следующие элементы строк. Команда movs не загружает регистр-аккумулятор во время пересылки.

В команду movs передаются адреса операндов. Только movs и еще одна строковая команда cmps работают с двумя операндами памяти. Все остальные команды требуют, чтобы один или оба операнда находились в одном из регистров микропроцессора. Команда movs, так же как и команды lods и stos, работает как с байтами, так и со словами.

Типы операндов, с которыми работают команды cmps, movs, scas, lods и stos, должны быть явно определены в программе. Оба операнда должны быть одного типа. Программист также может указать неявно тип операндов с помощью формата команды. Например, команда movsb используется для операций с байтами, а команда movsw — для операций со словами. Если в программе используется основная форма команды movs, то ассемблер проверяет переменные на правильность сегментной адресации и на совпадение типов.

Команда movs с префиксом rep дает эффективную команду пересылки блока. Имея счетчик символов в регистре ECX и указывающий направление пересылки флаг направления DF, команда rep movs пересылает данные из одного места памяти в другое очень быстро. Следующая программа (листинг 2.28) демонстрирует копирование одной строки в другую. Это уже знакомое нам консольное приложение.

Листинг 2.28. Программа, выполняющая копирование одной строки в другую

```
.386
.model flat, stdcall
  option casemap : none                     ; различаем регистр символов
  include \masm32\include\windows.inc
```

```
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib

.data
  conTitle      DB  "Copying of strings example", 0
  srcStr        DB  "This text is copied to dstStr", 0
  lenStr        EQU $-srcStr
  dstStr        DB  64 DUP (' ')
  readBuf       DB  ?
  lenReadBuf    DD  1
  hStdIn        DD  0
  hStdOut       DD  0
  chrsRead      DD  0
  chrsWritten   DD  0

  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11

.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex
  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex
  call    getout_hndl
  call    getinp_hndl

  ; копирование строки srcStr в dstStr

  mov     ECX, lenStr
  lea     ESI, srcStr
  lea     EDI, dstStr
```

```
next:
  lodsb
  stosb
  loop    next

  ; вывод содержимого строки dstStr в окно консоли

  push    EBX
  mov     EBX, offset dstStr
  mov     ECX, lenStr
  call    write_con
  pop     EBX

  ; ожидание ввода и выход из программы

  call    read_con
ex:
  push    0
  call    ExitProcess

;------ Процедуры -----

getout_hndl proc
  push    STD_OUTP_HNDL
  call    GetStdHandle
  mov     hStdOut, EAX
  ret
getout_hndl endp

getinp_hndl proc
  push    STD_INP_HNDL
  call    GetStdHandle
  mov     hStdIn, EAX
  ret
getinp_hndl endp

write_con proc
  push    0
  push    chrsWritten
```

```
  push    ECX
  push    EBX
  push    hStdOut
  call    WriteConsoleA
  ret
write_con endp

read_con proc
  push    0
  push    chrsRead
  push    lenReadBuf
  push    offset readBuf
  push    hStdIn
  call    ReadConsoleA
  ret
read_con endp
end start
```

Перед началом операции копирования необходимо установить флаг направления так, чтобы адреса источника и приемника увеличивались после каждой итерации. Для этого нужно установить флаг в положение `0` командой `cld`.

Операция копирования выполняется двумя командами: `lodsb` и `stosb`. Чтобы скопировать строку, необходимо организовать цикл, например при помощи команды `loop`. В регистре `ECX` содержится размер строки `srcStr`. Команда `lodsb` загружает байт из ячейки по адресу `ESI` (строка `srcStr`) в аккумулятор `AL`, а команда `stosb` записывает полученный байт из аккумулятора в ячейку памяти по адресу, содержащемуся в регистре `EDI` (строка `dstStr`). После операции чтения-записи содержимое регистров `ESI` и `EDI` автоматически инкрементируется на 1. Величина инкремента в этом случае определяется типом строковой команды. В программе копируются байты, поэтому и адреса будут увеличиваться на 1. Для того чтобы этот фрагмент кода отработал правильно, объем памяти, выделенный для строки-приемника (`dstStr`), должен быть по крайней мере не меньше размера строки-источника.

Окно работающего приложения с результатом копирования изображено на рис. 2.12.

Предыдущую программу можно упростить, если вместо двух команд `lodsb` и `stosb` использовать команду копирования строк `movsb` с префиксом повторения `rep` (листинг 2.29).

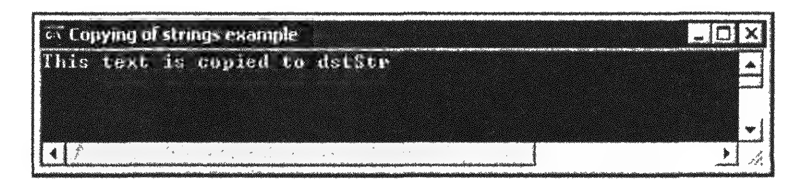


Рис. 2.12. Окно приложения, выполняющего операцию копирования одной строки в другую

Листинг 2.29. Фрагмент кода, использующий команду movsb для копирования одной строки в другую

```
  ...
srcStr        DB  "This text is copied to dstStr", 0
lenStr        EQU $-srcStr
dstStr        DB  64 DUP (' ')
  ...
cld
mov     ECX, lenStr
lea     ESI, srcStr
lea     EDI, dstStr
rep     movsb
  ...
```

Префикс rep использует в качестве параметра содержимое регистра ECX. В остальном же программа работает аналогично предыдущему примеру.

Операции копирования можно выполнять также и для массивов целых или вещественных чисел. В следующем фрагменте кода (листинг 2.30) содержимое целочисленного массива SARRAY копируется в массив DARRAY.

Листинг 2.30. Фрагмент кода, выполняющий копирование одного целочисленного массива в другой

```
  ...
SARRAY      DD 245, 11, -34, 56, 7, 19
LEN_SARRAY  DD ($-SARRAY)/4
DARRAY      DD 16 DUP (0)
  ...
cld
mov     ECX, DWORD PTR LEN_STR
```

```
lea     ESI, SARRAY
lea     EDI, DARRAY
rep     movsd
  ...
```

В этом фрагменте для копирования целых чисел используется команда movsd. Размер исходного массива задан переменной LEN_SARRAY. Выражение $-SARRAY определяет размер массива SARRAY в байтах, поэтому необходимо добавить деление на 4, чтобы получить размер в двойных словах.

Команда movs может использоваться для еще одной весьма полезной операции над двумя строками. Эта операция называется конкатенацией. При ее выполнении в конец строки-приемника помещаются элементы строки-источника. Откорректируем исходный текст предыдущего примера так, чтобы наше приложение выводило на экран содержимое строки-приемника после конкатенации (листинг 2.31).

Листинг 2.31. Конкатенация двух строк и вывод на экран содержимого строки-приемника

```
  ...
Src     DB "SOURCE STRING", 0
LenSrc  DD $-Src
Dst     DB "DEST STRING + ", 16 DUP (?)
LenDst  DD $-Dst-16
  ...
cld
mov     ECX, DWORD PTR LenSrc
lea     ESI, Src
lea     EDI, Dst
add     EDI, LenDst
rep     movsb

; вывод содержимого строки Dst на экран

push    EBX
mov     EBX, offset Dst
mov     ECX, LenDst
add     ECX, 16
call    write_con
pop     EBX
  ...
```

В этом фрагменте мы помещаем содержимое строки `Src` в конец строки `Dst`. Для этого необходимо зарезервировать необходимый объем памяти в строке `Dst`:

```
Dst     DB "DEST STRING + ", 16 DUP (?)
```

Кроме того, зададим начальное смещение в строке-приемнике на количество байт, занимаемое первыми символами:

```
lea     EDI, Dst
add     EDI, LenDst
```

Как обычно, поместим в регистр `ECX` число, равное количеству копируемых байт из строки-источника, и сбросим флаг направления в 0, чтобы индексы копирования увеличивались на каждой итерации. После выполнения этого фрагмента программы в `Dst` будет находиться строка:

```
"DEST STRING + SOURCE STRING"
```

На рис. 2.13 изображено окно работающего приложения.

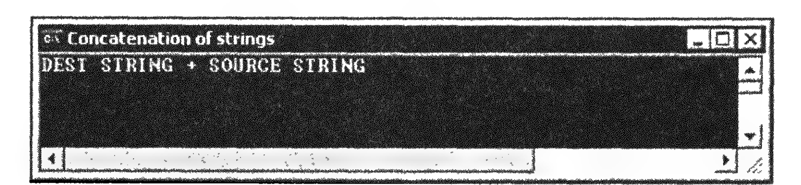


Рис. 2.13. Окно приложения, выполняющего конкатенацию двух строк

Конкатенация массивов целых или вещественных чисел немного отличается от аналогичной операции с символьными строками, хотя во многом они схожи. Принимающий массив должен иметь необходимое пространство для добавления новых элементов из массива-источника. Необходимое смещение в массиве-приемнике должно пересчитываться с учетом размерности в байтах элемента массива. Фрагмент программного кода, выполняющий конкатенацию двух массивов целых чисел представлен далее в листинге 2.32.

Листинг 2.32. Фрагмент кода, выполняющий конкатенацию двух массивов целых чисел

```
 ...
SARRAY  DD  498, -27, 31, -99, -36, 728, -20
LenS    EQU ($-SARRAY)/4
```

```
DARRAY  DD  11, 12, 13, 0, 0, 0, 0, 0, 0, 0
LenD    EQU ($-DARRAY)/4-7
  ...
cld
mov     ECX, DWORD PTR LenS
lea     ESI, SARRAY
lea     EDI, DARRAY
add     EDI, LenD*4
rep     movsd
  ...
```

В этом фрагменте кода элементы массива-источника SARRAY записываются в массив-приемник DARRAY, начиная с позиции четвертого элемента приемника. В регистры ESI и EDI мы помещаем, как обычно, адреса первых элементов этих массивов, а в регистр ECX — количество записываемых элементов. В данном случае регистр ECX содержит размер массива-источника SARRAY.

Поскольку необходимо поместить элементы источника в приемник начиная с четвертого элемента, то значение адреса в регистре EDI необходимо сместить на величину LenD, которая в нашем случае равна 12.

После выполнения операции копирования командой rep movsd массив DARRAY содержит элементы:

```
11 12 13 498 -27 31 -99 -36 728 -20
```

Другой распространенной операцией над строками и массивами является сравнение. Для сравнения элементов строк и массивов используется команда cmps и ее модификации. Следующий фрагмент программного кода (листинг 2.33) сравнивает две строки символов.

Листинг 2.33. Фрагмент кода, выполняющий сравнение двух строк

```
  ...
SRC     DB  "STRING 1 "
LSRC    EQU $-SRC
DST     DB  "STRING 1"
LDST    EQU $-DST
FLAG    DD  0
  ...
```

```
  cld
  lea     ESI, SRC
  lea     EDI, DST
  mov     ECX, LSRC
  mov     EDX, LDST
  cmp     ECX, EDX
  je      next_check
  jmp     continue
next_check:
  repe    cmpsb
  je      equal
  mov     EAX, FLAG
  jmp     continue
equal:
  mov     FLAG, 1
continue:
   ...
```

В этом фрагменте кода используется команда `cmpsb`, т. к. сравнение выполняется побайтно, с префиксом повторения `repe`. Если строки одинаковы, то переменной `FLAG` присваивается значение 1, а если не равны — то 0. В этом примере строки не равны, поэтому переменная `FLAG` будет сброшена в 0. Фрагмент программного кода для сравнения массивов целых чисел приведен в листинге 2.34.

Листинг 2.34. Фрагмент кода, выполняющий сравнение массивов целых чисел

```
   ...
  ISRC    DD  3, 16, 89, 11
  LISRC   EQU ($-ISRC)/4
  IDST    DD  3, 16, 89, 11, 9
  LIDST   EQU ($-IDST)/4
  FLAG    DD
   ...
  cld
  lea     ESI, ISRC
  lea     EDI, IDST
  mov     ECX, LISRC
  mov     EDX, LIDST
```

```
  cmp    ECX, EDX
  je     next_check
  jmp    continue
next_check:
  repe   cmpsd
  je     equal
  mov    EAX, FLAG
  jmp    continue
equal:
  mov    FLAG, 1
continue:
   ...
```

Различия в программных кодах для обработки массивов целых чисел и байтов связаны, прежде всего, с размерностью операндов. Поскольку целые числа занимают в памяти 4 байта, то вместо cmpsb необходимо использовать команду cmpsd для сравнения двойных слов. В регистр ECX по-прежнему заносим размер исходного массива, но теперь эта величина выражена количеством двойных слов. Вот почему мы делим полученные значения на 4.

```
ISRC    DD  3, 16, 89, 11
LISRC   EQU ($-ISRC)/4
IDST    DD  3, 16, 89, 11, 9
LIDST   EQU ($-IDST)/4
```

Еще один полезный пример — заполнение области памяти определенным символом или числом. Чтобы заполнить, например, символьную строку пробелами, можно написать код, приведенный в листинге 2.35.

Листинг 2.35. Фрагмент кода, заполняющего символьную строку пробелами

```
 ...
 SRC     DB  "Эта строка будет заполнена пробелами"
 LSRC    EQU $-SRC
 ...
 cld
 mov     AL, ' '
 mov     ECX, LSRC
 lea     EDI, SRC
 rep     stosb
 ...
```

Чтобы заполнить массив целых чисел нулями, необходимо использовать фрагмент кода, представленного в листинге 2.36.

Листинг 2.36. Фрагмент кода, заполняющего целочисленный массив нулями

```
  ...
ISRC    DD  3, 16, 89, 11, -99, 4
LISRC   EQU ($-ISRC)/4
  ...
cld
lea     EDI, ISRC
mov     ECX, LISRC
mov     EAX, 0
rep     stosd
  ...
```

Строковые команды ассемблера очень полезны для оптимизации программ, написанных на языках высокого уровня. Команды копирования строк, конкатенации, поиска элементов, сравнения строк и заполнения области памяти определенными значениями есть в любом языке высокого уровня. Ассемблерный вариант реализации таких команд, как правило, требует намного меньше программного кода и выполняется быстрее.

Рассмотрим еще один пример операции со строками. Очень часто требуется преобразовывать символы нижнего регистра клавиатуры в символы верхнего. В этом фрагменте кода использование строковых команд может неоправданно усложнить программу, поэтому будем использовать обычные операторы. Полностью исходный текст приложения выглядит так, как показано в листинге 2.37.

Листинг 2.37. Программа, переводящая символы из нижнего регистра в верхний

```
.386
.model flat, stdcall
  option casemap : none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\masm32.inc
  includelib \masm32\lib\user32.lib
```

```
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib

.data
  conTitle      DB  "Convert to upper", 0
  mes1          DB  "Before: ", 0
  len_mes1      EQU $-mes1
  mes2          DB  "After: ", 0
  len_mes2      EQU $-mes2

  S1            DB  "a string must be converted to upper!", 0dh, 0ah, 0
  LS1           EQU $-S1
  CharBuf       DB  "      ", 0
  len_charBuf   DD  $-charBuf
  readBuf       DB  ?
  lenReadBuf    DD  1

  hStdIn        DD  0
  hStdOut       DD  0

  chrsRead      DD  0
  chrsWritten   DD  0

  STD_INP_HNDL  DD -10
  STD_OUTP_HNDL DD -11

.code
start:
  call    AllocConsole
  test    EAX, EAX
  jz      ex
  push    offset conTitle
  call    SetConsoleTitleA
  test    EAX, EAX
  jz      ex
  call    getout_hndl
  call    getinp_hndl

  push    EBX
  mov     EBX, offset mes1
```

```
  mov     ECX, len_mes1
  call    write_con
  pop     EBX

  push    EBX
  mov     EBX, offset S1
  mov     ECX, LS1
  call    write_con
  pop     EBX

  push    EBX
  mov     EBX, offset mes2
  mov     ECX, len_mes2
  call    write_con
  pop     EBX

  ; преобразование символов из нижнего регистра в верхний

  lea     ESI, DWORD PTR S1
  mov     ECX, DWORD PTR LS1
next:
  mov     AL, BYTE PTR [ESI]
  cmp     AL, 'a'
  jb      next_addr
  cmp     AL, 'z'
  ja      next_addr
  and     AL, 0dfh
  mov     BYTE PTR [ESI], AL
next_addr:
  inc     ESI
  loop    next

  push    EBX
  mov     EBX, offset S1
  mov     ECX, LS1
  call    write_con
  pop     EBX

  ; ожидание ввода с консоли и выход
```

```
  call    read_con
ex:
  push    0
  call    ExitProcess

;------ Процедуры -------

getout_hndl proc
  push    STD_OUTP_HNDL
  call    GetStdHandle
  mov     hStdOut, EAX
  ret
getout_hndl endp

getinp_hndl proc
  push    STD_INP_HNDL
  call    GetStdHandle
  mov     hStdIn, EAX
  ret
getinp_hndl endp

write_con proc
  push    0
  push    chrsWritten
  push    ECX
  push    EBX
  push    hStdOut
  call    WriteConsoleA
  ret
write_con endp

read_con proc
  push    0
  push    chrsRead
  push    lenReadBuf
  push    offset readBuf
  push    hStdIn
  call    ReadConsoleA
```

```
  ret
read_con endp
end start
```

Перед началом преобразования загружаем в регистр ESI адрес строки, а в регистр ECX — ее размер. Поскольку мы имеем дело с литерами, то анализ выполняется для символов 'a'-'z', не затрагивая остальные. Сам алгоритм преобразования реализован в следующем фрагменте программного кода:

```
   ...
next:
  mov     AL, BYTE PTR [ESI]
  cmp     AL, 'a'
  jb      next_addr
  cmp     AL, 'z'
  ja      next_addr
  and     AL, 0dfh
  mov    BYTE PTR [ESI], AL
next_addr:
  inc     ESI
  loop    next
   ...
```

Окно работающего приложения изображено на рис. 2.14.

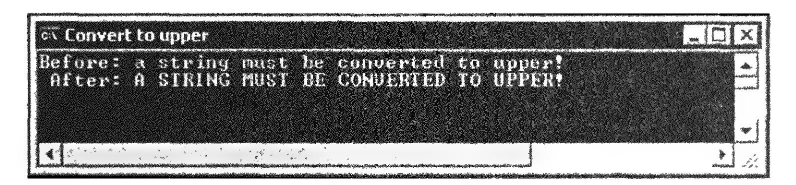


Рис. 2.14. Окно приложения, преобразующего символы нижнего регистра строки в символы верхнего регистра

Чтобы у читателя не сложилось впечатление, будто операции со строками можно эффективно выполнять только строковыми командами, приведем пример программы, где они вообще не используются. Операции над строками можно успешно выполнять и при помощи обычных команд ассемблера. Исходный текст консольного приложения на Delphi демонстрирует такой подход. В этой программе (листинг 2.38) выполняется сравнение двух строк и результат операции выводится на экран дисплея.

Листинг 2.38. Программа на Delphi, выполняющая сравнение двух строк с помощью обычных команд ассемблера

```
program cspas;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  s1, s2: PChar;
  flag: Boolean;

function cmpstring: Boolean; assembler;
asm
  mov     ESI, DWORD PTR s1
  mov     EDI, DWORD PTR s2
@again:
  mov     AL, BYTE PTR [ESI]
  mov     DL, BYTE PTR [EDI]
  push    EAX
  push    EDX
  xor     AL, DL
  pop     EDX
  pop     EAX
  jz      @streq
  jmp     @strnot_eq
@streq:
  test    AL, DL
  jz      @succ
  inc     ESI
  inc     EDI
  jmp     @again
@strnot_eq:
  mov     EAX, 0
  jmp     @quit
@succ:
  mov     EAX, 1
@quit:
end;
```

```
begin

  { TODO -oUser -cConsole Main : Insert code here }

  s1 := 'sTRING'#0;
  s2 := 'STRING'#0;
  WriteLn('s1: ', s1);
  WriteLn('s2: ', s2);
  Flag := cmpstring;
  if flag then
    WriteLn('Strings are equal !')
  else
    WriteLn('Strings are not equal !');
  ReadLn;
end.
```

Процедура `cmpstring` в самом начале загружает адрес строки-источника в регистр `ESI` и адрес строки-приемника в регистр `EDI`. Процедура сравнивает строки с завершающим нулем.

Элементы строк помещаются в регистры `AL` и `DL`, которые сравнивают их содержимое:

```
  ...
mov     AL, BYTE PTR [ESI]
mov     DL, BYTE PTR [EDI]
push    EAX
push    EDX
xor     AL, DL
pop     EDX
pop     EAX
jz      @streq
jmp     @strnot_eq
  ...
```

Если символы не равны, то происходит выход из процедуры с возвратом 0 в основную программу. Если символы равны, то процедура проверяет их на равенство 0 (команда перехода `jz @streq`):

```
  ...
```

```
@streq:
  test   AL, DL
  jz     @succ
  inc    ESI
  inc    EDI
  jmp    @again
   ...
```

Если элементы равны 0, то достигнут конец строки и сравнение прошло успешно, т. е. строки равны. В этом случае процедура возвращает в основную программу значение 1. Если же элементы не равны 0 (хотя и равны между собой), то выполняется переход на следующие адреса в строках и цикл сравнения повторяется.

В основной программе обе строки `s1` и `s2` определены как строки с завершающим нулем с помощью указателей типа `PChar`. В нашем случае строки не равны, поэтому в окно работающего приложения (рис. 2.15) будет выведено соответствующее сообщение.

```
F:\Program Files\Borland\Delphi7\Projects\PART2\CMP_STR_CON\csp...
s1: sTRING
s2: STRING
Strings are not equal!
```

Рис. 2.15. Окно приложения, выполняющего операцию сравнения двух строк

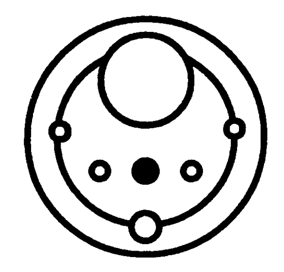
Как видите, при манипуляциях со строками можно обходиться и без специализированных команд, однако код получается несколько громоздким за счет дополнительных операций инкремента или декремента адресов и дополнительного анализа условий равенства символов и конца строк.

Самая высокая скорость выполнения строковых операций достигается обычно при копировании одной строки в другую или при перемещении элементов строки из одной области памяти в другую. Это особенно заметно при перемещении больших объемов данных.

Меньший выигрыш в производительности по сравнению с обычными командами дают команды поиска определенного элемента строки. На скорость выполнения строковых операций влияет и размерность операндов.

Мы рассмотрели только небольшую часть тех возможностей, которые предоставляет нам язык ассемблера в плане оптимизации обработки данных. В последующих главах мы будем использовать рассмотренный здесь материал и продемонстрируем дополнительные возможности ассемблера для улучшения качества программного кода.

Глава 3



Интерфейс с языками высокого уровня

Эта глава будет полезна как программистам, пишущим на языках высокого уровня, так и тем, кто предпочитает использовать для разработки приложений ассемблер. Программисты, пишущие на языках высокого уровня, применяют ассемблерные вставки и целые, отдельно скомпилированные модули для улучшения параметров своих приложений. Программисты-ассемблерщики часто используют в свою очередь логические структуры и выражения, свойственные языкам высокого уровня. Материал этой главы будет полезен и тем, и другим. Вначале мы рассмотрим наиболее часто используемые конструкции языков высокого уровня в ассемблерной интерпретации, а затем покажем, каким образом разработанные фрагменты программного кода можно объединять с программами на языках высокого уровня, используя для этого отдельно скомпилированные модули на ассемблере. Следует сказать, что все языки высокого уровня имеют, как правило, встроенный ассемблер, однако рассмотрение его использования для оптимизации программ отложим до *главы 6*.

3.1. Конструкции высокого уровня на языке ассемблера

Сначала рассмотрим оптимизацию программ на языках высокого уровня с помощью ассемблера, т. к. многие задачи успешно решаются с помощью ассемблерных процедур.

Анализ программ, написанных на языках высокого уровня, позволяет найти их слабые места, прежде всего, в нерациональном использовании инструкций выбора и циклических вычислений. Также значительно снижают производительность программ обработка больших массивов данных, строк и математические вычисления. Ни один компилятор ни в одном языке высокого уровня, как бы он ни оптимизировал программу по быстродействию или по размеру исполняемого модуля, не в состоянии устранить избыточность и неоптимальность кода. Это касается даже компилятора фирмы Intel с неплохими, казалось бы, характеристиками оптимизации на уровне команд процессора.

Наиболее легко поддаются оптимизации на ассемблере циклические вычисления, такие конструкции, как `if-else`, `while`, `repeat-until`, `case`. Как правило, оптимизация инструкций выбора и циклов основана на использовании *команд сравнения* и условных переходов в зависимости от результата сравнения. В общем виде это можно представить так:

```
   ...
  cmp     operand1, operand2
  Jcond   label1
  <операторы 1>
  jmp     label2
label1:
  <операторы 2>
label2:
   ...
```

Здесь `operand1` и `operand2` — переменные и/или выражения, `Jcond` — *оператор условного перехода* (`je`, `jl`, `jge`, `jz` или другой).

Любые, сколь угодно сложные конструкции языка высокого уровня можно представить в виде комбинаций операторов условных переходов и *операторов сравнения*, причем, несколькими способами. Далее мы рассмотрим варианты реализации конструкций языка высокого уровня в контексте практического применения. Начнем с инструкции выбора `if`.

3.1.1. Инструкция *if*

Одиночная инструкция `if` предназначена для выполнения команды или блока команд в зависимости от того, истинно или нет заданное условие.

На C++ такая инструкция будет иметь вид:

```
if (условие)
{
  <операторы>
}
```

На языке Pascal и, соответственно, в Delphi фигурные скобки заменяются операторами `begin` и `end`, а сама инструкция `if` будет выглядеть следующим образом:

```
if (условие) then
  begin
   <операторы>
  end
```

Более сложный вариант инструкции `if-else` позволяет выборочно выполнить одно из двух действий в зависимости от условия. Далее показан синтаксис инструкции на языке C++:

```
if (условие)
{
  <операторы 1>
}
else
{
  <операторы 2>
}
```

Вариант этой инструкции на Delphi будет выглядеть немного по-другому:

```
if (условие) then
  begin
   <операторы 1>
  end
else
  begin
   <операторы 2>
  end
```

В языке ассемблера нет таких конструкций, однако они довольно легко реализуются с помощью определенных последовательностей команд.

Рассмотрим, например, алгоритм проверки двух операндов на равенство и, в зависимости от результата, переход на ту или иную ветвь программы.

На языке Pascal и соответственно на Delphi такой алгоритм имеет вид:

```
if (operand1 = operand2) then
  begin
   ...
  end
else
  begin
   ...
  end
```

На Visual C++ .NET выражение будет выглядеть несколько иначе:

```
if (operand1 = operand2)
{
    ...
}
else
{
    ...
}
```

Язык ассемблера позволяет представить конструкцию `if-else` довольно просто:

```
  cmp     operand1, operand2
  je      EQUAL
  <команды 1>
  jmp     DIFF
EQUAL:
  <команды 2>
DIFF:
   ...
```

Возможен и другой вариант:

```
  cmp     operand1, operand2
  jne     DIFF
  <команды 2>
EXIT:
   ...
DIFF:
  <команды 1>
  jmp     EXIT
```

Теперь мы знаем достаточно об инструкции `if` и можем применить наши знания на практике.

Рассмотрим пример, в котором необходимо написать условие сравнения двух переменных целочисленного типа `X` и `Y`. В зависимости от результата

сравнения X принимает значение Y в случае, если X больше Y, и остается неизменным, если X меньше или равен Y.

Фрагмент кода на Delphi мог бы выглядеть так:

```
if X > Y then
   X := Y;
```

В C++ этот же фрагмент кода будет выглядеть следующим образом:

```
if (X > Y)
   X = Y;
```

Ассемблерная версия будет такой:

```
  mov     EAX, DWORD PTR Y
  cmp     EAX, DWORD PTR X
  jge     EXIT
  mov     DWORD PTR X, EAX
EXIT:
```

Проанализируем приведенный код.

Вспомним, что мы работаем с 32-разрядными операндами, поэтому все переменные и регистры описаны соответствующим образом. Команда сравнения cmp не работает с операндами, находящимися в оперативной памяти, поэтому один из операндов (в данном случае Y) сперва помещается в регистр EAX. Еще один важный момент: поскольку программы на языках высокого уровня работают в основном с переменными в памяти, очень удобно результат вычислений помещать сразу в переменную, например в X.

Решим такую задачу: необходимо найти сумму двух целых чисел X и Y при условии, что оба они находятся в диапазоне от 1 до 100.

В Delphi фрагмент программы с использованием инструкции if мог бы выглядеть так:

```
if (X <= 100) and (X >= 1) then
   if (Y <= 100) and (Y >= 1) then
      X := X + Y;
```

В Visual C++ этот фрагмент будет выглядеть следующим образом:

```
if ((X <= 100 && X >= 1) && (Y <= 100 && Y >= 1))
    X = X + Y
```

Программный код на ассемблере представлен в листинге 3.1.

Листинг 3.1. Фрагмент программы на ассемблере, использующей аналог инструкции `if` для сложения двух целых чисел

```
  cmp     DWORD PTR X, 1
  jge     check_x100
  jmp     EXIT
check_x100:
  cmp     DWORD PTR X, 100
  jle     check_y1
  jmp     EXIT
check_y1:
  cmp     DWORD PTR Y, 1
  jge     check_y100
  jmp     EXIT
check_y100:
  cmp     DWORD PTR Y, 100
  jg      EXIT
  mov     EAX, DWORD PTR Y
  add     DWORD PTR X, EAX
EXIT:
```

Как видно из алгоритма, для получения суммы чисел `X` и `Y` необходимо проанализировать как минимум четыре условия:

- `X` больше или равно `1`;
- `X` меньше или равно `100`;
- `Y` больше или равно `1`;
- `Y` меньше или равно `100`.

Только при одновременном выполнении всех четырех условий переменной `X` может быть присвоено значение суммы `X + Y`. Для решения такой задачи необходимо представить условие

```
(X <= 100 && X >= 1) && (Y <= 100 && Y >= 1)
```

в виде более простых конструкций. Это выражение распадается на четыре условия:

```
X <= 100, X >= 1, Y <= 100, Y >= 1
```

Задача упростилась. Каждое из полученных четырех условий легко проверяется с помощью команды cmp. Например, проверка X <= 100 и последующий переход выполняются так:

```
cmp     DWORD PTR X, 100
jle     check_y1
```

Остальные проверки можно выполнить с помощью аналогичных комбинаций команд.

Надо заметить, что ассемблерный аналог конструкций на языках высокого уровня может быть достаточно завуалирован и с первого взгляда неочевиден, что видно из следующего примера.

Предположим, необходимо найти модуль (абсолютное значение) целого числа X. Один из вариантов решения такой задачи — использование конструкции if-else.

Вот исходный текст программного кода для Delphi:

```
if X >= 0 then
   AbsX := X
else
   AbsX := -X;
```

где AbsX — переменная, в которой хранится модуль X.

Вариант использования конструкции if-else на Visual C++:

```
if (X >= 0)
   AbsX = X
else
   AbsX = -X
```

Ассемблерная реализация такой конструкции выглядит следующим образом:

```
cmp     DWORD PTR X, 0
jl      NOT_X
jmp     EXIT
```

```
NOT_X:
  neg     DWORD PTR X
EXIT:
  mov     EAX, DWORD PTR X
  mov     DWORD PTR AbsX, EAX
```

И в ветви `if`, и в ветви `else` выполняется *оператор присваивания*. По смыслу можно объединить эти два присваивания, поместив их в конец фрагмента кода:

```
  mov     EAX, DWORD PTR X
  mov     DWORD PTR AbsX, EAX
```

Ветвь `else` представлена на ассемблере командой:

```
NOT_X:
  neg     DWORD PTR X
```

Результат сохраним в переменной `AbsX`. Это далеко не единственный вариант реализации этого фрагмента программы. Думаю, читатель при желании сможет разработать свой вариант решения.

3.1.2. Цикл *while*

Этот цикл используется в тех случаях, когда число повторений цикла заранее неизвестно. Цикл `while` — это цикл с предварительным условием, и его выполнение или невыполнение зависит от начальных условий. Синтаксис этого выражения можно представить в общем виде следующим образом:

```
while (условие)
    <операторы цикла>.
```

Выход из цикла осуществляется, если условие окажется ложным. Так как истинность условия определяется в начале каждой итерации, вполне может оказаться, что тело цикла не выполнится ни разу.

В C++ цикл `while` имеет вид:

```
while (условие)
{
  <операторы цикла>
}
```

Синтаксис этой конструкции в Delphi несколько иной:

```
while <условие> do
begin
  <операторы цикла>
end
```

Рассмотрим пример, где используется цикл `while`. Пусть имеется массив из 10 целых чисел. Требуется найти количество элементов массива, предшествующих первому встретившемуся нулевому элементу (если таковой будет найден). Фрагмент программы должен вернуть число элементов, предшествующих первому нулевому, или `0`, если нулевой элемент не найден. Такой фрагмент кода можно применить для поиска и выделения строк с завершающим нулем. Подобная задача легко решается при помощи цикла `for`. Но мы решим ее, используя цикл `while`.

Вначале, как обычно, представим исходный текст программы на языке высокого уровня. Фрагмент кода на Delphi приведен в листинге 3.2.

Листинг 3.2. Пример кода на Delphi, выполняющий подсчет ненулевых символов в массиве с использованием цикла `while`

```
  ...
var
 X1: array[1..10] of Integer = (12, 90, -6, 30, 22, 10, 22, 89, -0, 47);
 IX1: Integer;
 SX1: Integer;
 Counter: Integer;

  ...

 Counter := 0;
 IX1 := 1;
 SX1 := SizeOf(X1) div 4;

 while (X1[IX1] <> 0) do
 begin
   Inc(Counter);
   if (IX1 = SX1) then break;
   Inc(IX1);
 end;
```

```
if (Counter = SX1) then
  Counter := 0;
 ...
```

Здесь X1 — массив целых чисел, IX1 — текущий индекс массива, SX1 — размер массива, Counter — счетчик элементов.

Этот фрагмент кода выполняется следующим образом:

- после инициализации переменных программа в начале каждой итерации цикла while проверяет неравенство элемента массива X1 на 0. Если элемент равен 0, происходит немедленный выход из цикла;
- если условие верно, т. е. элемент массива не равен 0, выполняется тело цикла. При этом инкрементируются счетчик Counter и индекс массива IX1. Если обнаружен последний элемент массива, происходит выход из цикла (инструкция if);
- в любом случае счетчик Counter содержит количество элементов, предшествующих первому нулевому, или 0, если нулевой элемент вообще не обнаружен.

Программный код в Visual C++ для этой же задачи представлен в листинге 3.3.

Листинг 3.3. Пример кода на Visual C++ с использованием цикла while

```
 ...
int   X1[10] = {12, 90, -6, 30, 22, 10, 22, 89, -0, 47};
int   Counter = 0;
int   IX1 = 0;
int   SX1 = sizeof (X1);
while (X1[IX1] != 0)
{
  Counter++;
  if (IX1 == SX1) break;
  IX1++;
};
if (Counter == SX1)
  Counter = 0;
 ...
```

Вариант решения задачи на языке ассемблера (листинг 3.4) выглядит на первый взгляд более сложным, чем в предыдущих примерах.

Листинг 3.4. Пример кода на ассемблере с использованием цикла `while`

```
.386
.model flat, stdcall
.data
  X1      DD 2, -23, 5, 9, -1, 0, 9, 3
  SX1     DD $-X1
  IX1     DD 1
  Counter DD 0
.code
start:
  push    EBX
  mov     ECX, 0
  mov     EBX, offset X1
  mov     EDX, DWORD PTR SX1
  shr     EDX, 2
AGAIN:
  mov     EAX, DWORD PTR [EBX]
  cmp     EAX, 0
  je      RUNOUT
  inc     ECX
  cmp     ECX, EDX
  je      RUNOUT
  add     EBX, 4
  jmp     AGAIN
RUNOUT:
  cmp     ECX, EDX
  jne     SET_CNT
  xor     ECX, ECX
SET_CNT:
  mov     DWORD PTR Counter, ECX
  pop     EBX
   ...
end start
```

Необходимо сделать несколько важных замечаний. Первое касается использования регистров. При работе с внешними программами и модулями на языках высокого уровня всегда старайтесь сохранить регистры EBX,

языках высокого уровня всегда старайтесь сохранить регистры EBX, EBP, ESI и EDI в стеке. Что касается остальных регистров (EAX, ECX и EDX), то вы можете использовать их по своему усмотрению.

Второе замечание касается работы с массивами данных и строками на ассемблере. В операционной системе Windows для доступа к таким данным всегда используются 32-разрядные переменные, которые хранят адреса массивов или строк. Поэтому для осуществления доступа к элементам массива X1 необходимо поместить его адрес в регистр EBX:

```
mov     EBX, offset X1
```

Для работы нам понадобится и размер массива, который мы сохраним в регистре EDX:

```
mov     EDX, DWORD PTR SX1
```

Счетчик ненулевых элементов мы поместим в регистр ECX. Так как каждый элемент массива занимает в памяти 4 байта (двойное слово), то для доступа к последующему элементу мы используем команду

```
add     EBX, 4
```

В нашей задаче присутствуют две структуры высокого уровня — цикл while и оператор условия if. Цикл while реализован с помощью трех операторов:

```
mov     EAX, DWORD PTR [EBX]
cmp     EAX, 0
je      RUNOUT,
```

а условие if — операторами:

```
cmp     ECX, EDX
je      RUNOUT
```

Если нулевой элемент вообще не найден, то в счетчик ненулевых элементов по условию задачи записываем 0:

```
cmp     ECX, EDX
jne     SET_CNT
xor     ECX, ECX
```

Предыдущий фрагмент кода на ассемблере из трех операторов эквивалентен оператору на Delphi:

```
if Counter = SX1 then
  Counter := 0;
```

Мы так подробно остановились на ассемблерном варианте программы для того, чтобы читатель понял, что однозначного решения для оптимизации логических структур в языках высокого уровня не существует! "Строительным" кирпичиком такой оптимизации является все та же пара команд ассемблера:

```
cmp    operand1, operand2
Jcond  label
```

В принципе, на ассемблере можно реализовать сколь угодно сложные логические выражения и ветвления. Все ограничивается только фантазией и опытом разработчика.

3.1.3. Цикл *repeat-until (do-while)*

Операторы цикла `repeat-until` (Pascal, Delphi) и `do-while` (C, C++) организуют выполнение цикла, состоящего из любого числа операторов с заранее неизвестным числом повторений. Тело цикла в любом случае будет выполнено хотя бы один раз. Выход из цикла происходит, когда становится истинным некоторое логическое условие. Цикл `repeat-until` можно представить в виде конструкции на языке Delphi:

```
repeat
  <операторы>
until <условие>
```

По аналогии на языке C++ цикл `do-while` имеет вид:

```
do
{
  <операторы>
}
while <условие>
```

Приведем примеры реализации таких циклов на языках высокого уровня. Как всегда, решим при этом практически полезные задачи.

В следующем примере требуется найти сумму первых четырех элементов массива целых чисел. Пусть размерность массива равна 7.

Вариант программного кода в Delphi представлен в листинге 3.5.

Листинг 3.5. Фрагмент кода на Delphi, в котором находится сумма первых четырех элементов массива

```
   ...
var
  X1:  array [1..7] of Integer = (2, -4, 5, 1, -1, 9, 3);
  IX1: Integer;
  sumX1: Integer;

   ...
  IX1 := 1;
  sumX1 := 0;

  repeat
    sumX1 := sumX1 + X1[IX1];
    IX1 := IX1+1;
  until (IX1 > 4);
   ...
```

Фрагмент кода довольно прост и в пояснениях не нуждается. На Visual C++ такая программа выглядит так, как представлено в листинге 3.6.

Листинг 3.6. Фрагмент кода на C++, в котором находится сумма первых четырех элементов массива

```
   ...
  int X1[7] = {2, -4, 5, 1, -1, 9, 3};
  int IX1 = 0;
  int sumX1 = 0;
  do
  {
    sumX1 = sumX1 + X1[IX1];
    IX1++;
  }
```

```
while (IX1 <= 3);
 ...
```

В ассемблерном варианте программы цикл `repeat-until` может быть реализован при помощи операций сравнения и условных переходов. Проверка условия (в нашем случае текущего индекса массива) осуществляется после выполнения операторов тела цикла.

В листинге 3.7 представлен исходный текст программы на ассемблере.

Листинг 3.7. Фрагмент кода на ассемблере, в котором находится сумма первых четырех элементов массива

```
.386
.model flat, stdcall
.data
  X1     DD  2, -23, 5, 9, -1, 9, 3
  SX1    DD  $-X1
  IX1    DD  1
  CNT    EQU 3
  SUMX1  DD  0

.code
start:
  push    EBX
  mov     EBX, offset X1
  mov     EAX, 0
  mov     EDX, DWORD PTR SX1
  shr     EDX, 2
  cmp     EDX, CNT
  jl      EXIT

NEXT:
  add     EAX, [EBX]
  cmp     DWORD PTR IX1, CNT
  jg      EXIT
  inc     DWORD PTR IX1
  add     EBX, 4
  jmp     NEXT
```

```
EXIT:
  mov     DWORD PTR SUMX1, EAX
  pop     EBX
   ...
end start
```

Вначале инициализируем все необходимые переменные. Для доступа к элементам массива его адрес помещаем в регистр EBX:

```
mov     EBX, offset X1
```

Начальное значение суммы, равное 0, помещаем в регистр EAX:

```
mov     EAX, 0
```

Условие ассемблерного цикла repeat-until проверяется командой:

```
cmp     DWORD PTR IX1, CNT
```

где IX1 — текущий индекс массива.

Поскольку целочисленное значение элемента массива занимает в памяти двойное слово, то, как и в предыдущем примере, для доступа к последующему элементу мы должны увеличивать значение адреса на 4:

```
add     EBX, 4
```

Результат помещается в переменную SUMX1 для дальнейшего использования.

3.1.4. Цикл *for*

Оператор цикла for организует выполнение оператора или группы операторов определенное число раз.

В общем виде цикл можно представить так:

```
for(выражение-инициализатор; условие; выражение-модификатор)
    <операторы>
```

Дойдя до цикла, программа сразу выполняет выражение-инициализатор, которое устанавливает начальное значение счетчика цикла. Затем анализируется условие, которое называется еще условием прекращения цикла. Пока

оно истинно, цикл будет выполняться. Каждый раз после прохождения тела цикла выражение-модификатор изменяет счетчик цикла. Если проверка условного выражения дает `FALSE`, происходит выход из цикла и выполняются операторы, непосредственно следующие за `for`.

Представление оператора в языках Delphi и C++ отличается. Вначале рассмотрим конструкцию цикла `for` в Delphi. Имеются два варианта реализации этого цикла в зависимости от направления изменения модификатора цикла:

```
for Counter := First to Last          // для случая, когда Last > First
for Counter := First downto Last      // для случая, когда Last < First
```

В Visual C++ цикл `for` имеет единое представление независимо от направления изменения модификатора:

```
for (инициализатор; условие; модификатор)
```

Чаще всего цикл `for` используется для математических вычислений итерационного типа с постоянным приращением на каждой итерации с заранее известным числом повторений или для поиска элементов массивов или строк. Продемонстрируем применение цикла `for` на следующем примере. Пусть требуется найти сумму первых 7 элементов массива, состоящего из 10 целых чисел.

Вариант программного кода для этой задачи в Delphi представлен в листинге 3.8.

Листинг 3.8. Фрагмент кода на Delphi с использованием цикла `for`

```
  ...
var
  X1: array[1..10] of Integer = (4, -6, 43, -5, 22, -78, 90, 56, 1, -43);
  IX1: Integer;
  SX1: Integer;
  SUMX1: Integer;
begin
  SUMX1 := 0;
  SX1 := SizeOf(X1) div 4;
  for IX1 := 1 to SX1 do
    SUMX1 := SUMX1+X1[IX1];
end;
  ...
```

В C++ .NET тот же фрагмент кода мог бы выглядеть так, как представлено в листинге 3.9.

Листинг 3.9. Фрагмент кода на C++ с использованием цикла `for`

```
  ...
int i1[10] = {3, -5, 2, 7, -9, 1, -3, -7, -11, 15};
int isum = 0;
for (int cnt = 0; cnt < 7; cnt++)
{
  isum = isum + i1[cnt];
}
  ...
```

Цикл `for` на ассемблере удобно реализовать с помощью команды `loop`. В этом случае переменная цикла помещается в регистр `ECX`. Фрагмент кода для нахождения суммы первых семи элементов массива на ассемблере приведен в листинге 3.10.

Листинг 3.10. Фрагмент кода на ассемблере, реализующий цикл `for`

```
   ...
.data
  i1      DD 3, -5, 2, 7, -9, 1, -3, -7, -11, 15
  isum    DD 0
.code
start:
  mov     ECX, 7
  xor     EAX, EAX
  lea     ESI, DWORD PTR i1
next:
  add     EAX, [ESI]
  add     ESI, 4
  loop    next
  mov     DWORD PTR isum, EAX
   ...
end start
```

3.1.5. Условный оператор *case*

Условный оператор `case` позволяет осуществить выбор одного варианта из нескольких возможных без использования конструкций `if...else`. В операторе `case` проверяемая переменная обязана принадлежать к перечисляемому типу. Использование других типов не допускается. Оператор `case` широко используется в Pascal (Delphi) для организации множественных ветвлений.

Структуру этого оператора можно представить в виде:

```
case N of
     N1: <оператор 1>
     N2: <оператор 2>
      ...
     NN: <оператор N>
      ...
```

Чтобы реализовать оператор `case` на ассемблере, можно использовать сравнение для каждого случая с переходом на соответствующую метку в программе (листинг 3.11).

Листинг 3.11. Фрагмент кода на ассемблере, реализующий оператор case

```
   ...
  mov     EAX, DWORD PTR N
  cmp     EAX, VALUE_1
  je      BRANCH_1
  cmp     EAX, VALUE_2
  je      BRANCH_2
  cmp     EAX, VALUE_3
  je      BRANCH_3
   ...
  cmp     EAX, VALUE_N
  je      BRANCH_N
   ...

BRANCH_1:
  <операторы>
BRANCH_2:
  <операторы>
```

```
BRANCH_N:
  <операторы>
   ...
```

Часто бывает удобно вместо условных переходов сразу вызывать подпрограммы-обработчики условия (листинг 3.12).

Листинг 3.12. Фрагмент кода на ассемблере, использующий подпрограммы-обработчики условия

```
   ...
  mov      EAX, DWORD PTR N
  cmp      EAX, VALUE_1
  jne      BRANCH_1
  call     PROC_1
  jmp      EXIT

BRANCH_1:
  cmp      EAX, VALUE_2
  jne      BRANCH_2
  call     PROC_2
  jmp      EXIT

BRANCH_2:
  cmp      EAX, VALUE_3
  jne      BRANCH_3
  call     PROC_3
  jmp      EXIT

BRANCH_3:
  cmp      EAX, VALUE_4
  jne      EXIT
  call     PROC_4
   ...
EXIT:
  cmp     EAX, VALUE_2
  je      BRANCH_2
  cmp     EAX, VALUE_3
  je      BRANCH_3
```

```
  ...
cmp    EAX, VALUE_N
je     BRANCH_N
  ...
```

3.2. Общие принципы построения интерфейсов с языками высокого уровня

Рассмотрим наиболее общие вопросы, касающиеся построения интерфейсов при вызове процедур на ассемблере из программ, написанных на языках высокого уровня. При обдумывании, как лучше всего изложить данный материал, автором были сделаны некоторые выводы, касающиеся выбора средств программирования как на языках высокого уровня, так и на ассемблере.

В основном программисты пишут на одном из двух языков высокого уровня: Pascal или C++. Правда, встречаются и универсалы, работающие в этих двух языках одновременно. Современные средства быстрого проектирования с использованием C++ и Pascal, такие как Microsoft Visual C++ .NET, Borland C++ Builder 6 и Borland Delphi 7, пользуются наибольшей популярностью среди программистов. Исходя из этого автор решил использовать для демонстрации основных принципов построения интерфейсов две среды программирования двух наиболее ярких представителей (и конкурентов) на рынке программных средств проектирования — Microsoft Visual C++ .NET и Borland Delphi 7.

Не стоит отдавать предпочтение ни одному из этих программных продуктов: и у Microsoft Visual C++ и у Borland Delphi есть свои сильные и слабые стороны. Это прекрасные средства разработки, и спор по поводу того, что лучше — C++ или Pascal — в настоящее время не имеет смысла: можно писать хорошие программы, используя любое из этих средств разработки в зависимости от собственных предпочтений.

Мы не будем использовать в книге более ранние версии продуктов этих фирм, т. к. они не соответствуют современным подходам и предназначены в основном для разработки устаревших 16-разрядных приложений.

Все примеры программного кода написаны в двух исполнениях — для Microsoft Visual C++ .NET и Borland Delphi 7. Каждый пример состоит из основной программы на языке высокого уровня (C++ и/или Delphi) и ассемблерной процедуры, вызываемой этой программой.

Для разработки примеров используют разные типы шаблонов приложений — от классического процедурно-ориентированного Windows-приложения до SDI-приложения. Все примеры рассчитаны на то, чтобы показать практическую реализацию различных типов интерфейсов языков высокого уровня и ассемблера.

Большинство примеров являются оригинальными разработками автора и более нигде не встречаются. Поэтому везде, где необходимо, даются подробные комментарии, особенно для программ на языках высокого уровня. К сожалению, мы не сможем подробно рассмотреть все аспекты программирования в Visual C++ .NET и Delphi 7, однако многие сведения, необходимые для практической работы, приводятся по ходу описания программного кода.

Программные модули на ассемблере мы будем разрабатывать с использованием компиляторов TASM 5.0 фирмы Borland и MASM 6.14 фирмы Microsoft. Примеры исходных текстов программ будем приводить как для TASM 5.0, так и для MASM 6.14. Каждый программист предпочитает работать с определенными инструментами при написании программ. Кто-то программирует в Delphi и предпочитает работать с TASM 5.0, кто-то работает в Visual C++ и предпочитает MASM 6.14. Большинство же разработчиков (в том числе и автор этих строк) применяют смешанные модели для разработки приложений.

К сожалению, война стандартов между Microsoft и Borland вынуждает приводить примеры интерфейсов с языками высокого уровня для двух этих компиляторов по отдельности. Тем не менее постарайтесь минимизировать все изменения и доработки программного кода на ассемблере при переходе от TASM 5.0 к MASM 6.14 и наоборот.

Для написания наших ассемблерных модулей будем использовать упрощенный синтаксис языков ассемблера. Это значит, что везде в исходных текстах будут использоваться директивы `.data` и `.code`. Не будем описывать здесь все опции компиляторов и компоновщиков MASM и TASM. В наших программах, как правило, мы будем использовать несколько таких опций, и разъяснения будут приводиться по ходу текста. Высокоуровневые конструкции языков ассемблера мы также не будем здесь использовать — они экономят место, но затрудняют анализ программ.

В примерах этой главы мы будем использовать отдельно скомпилированные модули на ассемблере, которые будем компоновать с программами на C++ .NET и Delphi 7. В общем случае командная строка для компилятора TASM выглядит так:

```
tasm32 /ml <имя_файла.asm> <имя_файла.obj>
```

Если используется MASM, командная строка для компилятора будет выглядеть следующим образом:

```
ml /c /Fo <имя_файла.obj> <имя_файла.asm>
```

Никакие дополнительные опции компиляторов для получения файлов с расширением .OBJ не нужны.

Теперь поговорим более подробно о разработке интерфейса с языками высокого уровня. При решении этой задачи программист должен учитывать следующее:

- правила согласования имен идентификаторов (переменных и процедур), помещенных в объектные файлы. Компилятор языка высокого уровня может изменять или нет оригинальные имена в объектном модуле, поэтому важно знать, происходит ли такое изменение и как;
- модель памяти, используемую ассемблерным модулем (`tiny`, `small`, `compact`, `medium`, `huge`, `large` или `flat`);
- параметры вызова нашей подпрограммы на ассемблере. Параметры вызова — это довольно обширное понятие и включает следующие аспекты, которые должен принимать во внимание программист:
  - нужно ли сохранять регистры в подпрограмме? Если да, то какие;
  - порядок передачи параметров вызываемой подпрограмме;
  - метод передачи параметров в подпрограмму (с использованием регистров, стека, разделяемой памяти);
  - способ передачи параметров в вызываемую подпрограмму (по значению или по ссылке);
  - если передача параметров подпрограмме осуществляется через стек, то как должен восстанавливаться указатель стека — вызывающей или вызываемой подпрограммой;
  - метод возвращения значения в вызывающую подпрограмму (через стек, регистры или общую область памяти).

Рассмотрим все эти вопросы более подробно. Начнем с согласования имен идентификаторов. Поскольку мы используем Delphi и Visual C++, то, соответственно, будем рассматривать технологии работы с внешними подпрограммами применительно к языкам Pascal и C++. С языком Pascal все просто. Все строчные буквы в именах внешних идентификаторов преобразуются в прописные. Компилятор C++ не изменяет регистра букв, но имена идентификаторов по этой причине считаются чувствительными к регистру. Кроме того, компилятор C++ перед всеми внешними именами помещает префикс в виде символа подчеркивания.

Следующий момент — модели памяти, используемые внешними подпрограммами. Для 32-разрядных приложений используется только одна модель памяти — `flat`. Она поддерживается как компилятором Pascal, так и C++.

Что касается параметров вызова внешних подпрограмм, то, несмотря на сложность и некоторую запутанность описания во многих литературных источниках, в реальной жизни все оказывается намного проще. Для программиста, желающего разобраться с этими нагромождениями директив и со-

глашений, хочется выделить основные моменты процесса вызова внешних процедур из языков высокого уровня.

- Для 32-разрядных приложений параметры в вызываемую процедуру передаются одним из двух способов: либо по значению, либо по ссылке. При передаче параметра по значению в процедуру передается непосредственно сам 32-разрядный операнд, а при передаче по ссылке — адрес (тоже 32-разрядное значение) этого операнда.
- Все параметры являются 32-разрядными. Понятия "ближняя ссылка" и "дальняя ссылка" не различаются! Все ссылки в адресном пространстве 32-разрядных приложений являются "ближними". Например, не имеет смысла объявлять подпрограммы как NEAR или FAR:

  ```
  MyProc    PROC NEAR
  ```

  или

  ```
  MyProc    PROC FAR
  ```

  поскольку компилятор все равно интерпретирует все вызовы как ближние.
- Параметры в вызываемую процедуру передаются через стек, через регистры или через общую область памяти. Передача параметров через общую область памяти для 32-разрядных приложений — вещь довольно экзотическая из-за сложности реализации и применяется обычно системными программистами и разработчиками драйверов устройств. Это отдельная тема, и мы ее рассматривать не будем. Все варианты передачи параметров через стек или регистры представлены в табл. 3.1.

Таблица 3.1. Варианты передачи параметров

Директива	Порядок передачи параметров	Освобождение стека	Передача параметров через регистры
register, fastcall	Слева направо	Процедура	EAX, EDX, ECX
pascal	Слева направо	Процедура	Нет
cdecl	Справа налево	Вызывающая программа	Нет
stdcall	Справа налево	Процедура	Нет
safecall	Справа налево	Процедура	Нет

Хотелось бы сделать некоторые пояснения, касающиеся табл. 3.1. Порядок передачи параметров для каждой директивы указывает компилятору, каким образом параметры передаются в вызываемую процедуру. Для директив

pascal, cdecl, stdcall и safecall параметры передаются через стек, а при использовании директив register или fastcall — через регистры.

Перед возвращением в основную программу необходимо восстановить указатель стека. Это относится к директивам pascal, cdecl, stdcall и safecall. Что касается применения тех или иных способов вызова внешних процедур, то здесь не существует однозначных рецептов. Если вы работаете с API функциями Windows, то для них стандартным способом вызова является stdcall или safecall. Директиву cdecl лучше использовать для вызова процедур и функций из программ, написанных в C++.

Наиболее быстрым способом передачи параметров является регистровый (register, или fastcall). Директива register используется в большинстве языков высокого уровня, но разработчики Microsoft решили назвать ее по-другому, и в Visual C++ она определяется иначе — fastcall. Стек в этом случае не используется, поэтому мы получаем выигрыш по скорости выполнения подпрограммы.

Директива pascal используется редко и только в целях *обратной совместимости* (backward compatibility). Мы не будем использовать ее в наших примерах.

Все подпрограммы возвращают результат (значение или адрес) в регистре EAX.

Для иллюстрации вышесказанного разработаем небольшую демонстрационную процедуру на ассемблере, которая будет находить сумму двух целых чисел. Наша процедура будет работать с приложениями, написанными на Delphi 7 и Visual C++ .NET. Программа на языке высокого уровня будет передавать ей два целочисленных аргумента в качестве параметров, а в качестве результата получать сумму этих чисел.

На этом простом примере я покажу, как работает интерфейс ассемблерных процедур и программ на языках высокого уровня для разных вариантов передачи параметров и с двумя несколько отличными друг от друга инструментами разработки — Delphi и Visual C++. Для компиляции исходного текста процедуры на ассемблере мы будем использовать как MASM 6.14, так и TASM 5.0.

Каркас процедуры для варианта передачи параметров stdcall приведен в листинге 3.13.

Листинг 3.13. Процедура на ассемблере, выполняющая суммирование двух чисел

```
.386
.model flat
  public AddInts
```

```
.data
.code
AddInts proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, DWORD PTR [EBP+8]
  add     EAX, DWORD PTR [EBP+12]
  pop     EBP
  ret     8
AddInts endp
end
```

Процедура получает два целочисленных параметра через стек, а результат возвращает в регистре EAX.

Первые две строки — стандартное начало для 32-разрядных приложений. Далее, поскольку процедура AddInts должна быть доступна из внешних модулей, необходимо объявить ее с директивой public. Первые две строки тела процедуры:

```
push    EBP
mov     EBP, ESP
```

необходимы для доступа к параметрам в стеке с помощью регистра EBP. Сами параметры находятся в стеке по адресам [EBP+8] и [EBP+12]. Но где из них первый, а где второй? Чтобы ответить на этот вопрос, необходимо указать в вызывающей программе порядок передачи аргументов.

Посмотрим теперь, как наша процедура будет вызываться из Delphi и C++. Начнем с Delphi. Фрагмент кода для этого случая приведен в листинге 3.14.

Листинг 3.14. Вызов ассемблерной процедуры из программы на Delphi

```
implementation
  {$R *.dfm}
  {$L ADDINTS.OBJ}
function AddInts(X1: Integer; X2: Integer): Integer; stdcall; external;
   ...
var
  X1, X2: Integer;
  SUM: Integer;
```

```
begin
  X1 := 23;
  X2 := -67;
  SUM := AddInts(X1, X2);
   ...
```

Строка `{$L ADDINTS.OBJ}` указывает компилятору и компоновщику на то, что будет использован внешний объектный файл. Строка

```
function AddInts(X1: Integer; X2: Integer): Integer; stdcall; external;
```

определяет поведение вызывающей процедуры.

Во-первых, директива `stdcall` (см. табл. 3.1) указывает на то, что параметры `X1` и `X2` передаются через стек справа налево, т. е. первым в стек помещается `X2`, затем `X1`. Поскольку стек растет от больших адресов памяти к меньшим, то `X2` будет размещаться по большему адресу, а `X1` — по меньшему.

После вызова процедуры `AddInts` и сохранения регистра `EBP` в стеке расположение параметров `X1` и `X2` в стеке будет выглядеть так, как изображено на рис. 3.1.

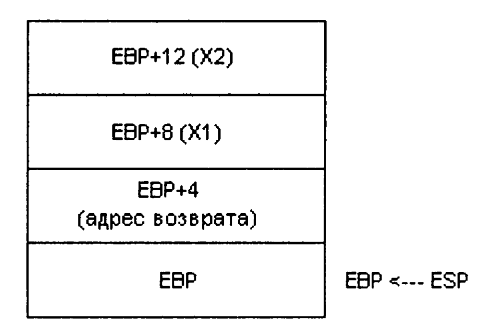


Рис.3.1. Расположение параметров в стеке

Для доступа к параметрам в стеке и их суммирования, как следует из рис. 3.1, можно использовать команды процедуры:

```
mov    EAX, DWORD PTR [EBP+8]
add    EAX, DWORD PTR [EBP+12]
```

Директива `external` объявляет процедуру `AddInts` внешней, т. е. расположенной в другом модуле. Ключевое слово `function` указывает на то, что

процедура возвращает значение в вызывающую программу. Думается, не возникнет путаницы с определением процедуры или функции в языках высокого уровня и употреблением этого термина в тексте книги.

Параметры x1 и x2 являются целыми переменными, и в процедуру AddInts передаются их значения. Это видно из определения x1 и x2 в секции var объявления переменных.

Результат сложения, как видно из исходного текста процедуры AddInts, возвращается в регистре EAX. Возвращая управление основной программе, процедура AddInts в соответствии с директивой stdcall должна сама восстановить стек. Перед последней командой ret там находятся два двойных слова, т. е. 8 байт. Чтобы удалить их из стека, необходимо в команде ret указать параметр 8. Можно использовать вместо ret 8 последовательность команд:

```
add     ESP, 8
ret
```

Сохраним исходный текст нашей программы в файле AddInts.asm. Имя исходного файла никак не связано с нашей процедурой, и выбрали мы его только для удобства.

Далее необходимо откомпилировать наш ASM-файл. Командная строка для компилятора Borland TASM 5.0 будет выглядеть так:

```
tasm32 /ml AddInts.asm
```

Параметр ml вынуждает компилятор различать регистр символов. Для компилятора MASM 6.14 командная строка будет выглядеть иначе:

```
ml /c AddInts.asm
```

Опция /c указывает компилятору, что необходима только трансляция исходного модуля, что нам и нужно для получения файла объектного модуля. Если компиляция исходного модуля прошла успешно, то мы получим файл AddInts.obj, с которым и будем далее работать. Не забудем скопировать наш объектный модуль в рабочий каталог Delphi-приложения перед компоновкой всего приложения.

Рассмотрим, как изменится наша процедура и ее вызов в среде программирования Visual C++ .NET. Напомним, что мы работаем с вызываемой процедурой, используя директиву stdcall. Внесем коррективы в исходный текст нашей процедуры AddInts.

Необходимо добавить в имени вызываемой процедуры символ подчеркивания и суффикс `@n`, где `n` — число байт, требуемое для передачи параметров. В данном случае `n` равно `8`. Такая форма именования процедуры отвечает требованиям компилятора C++ для корректной работы. С учетом этих изменений исходный текст будет выглядеть так, как показано в листинге 3.15.

Листинг 3.15. Процедура на ассемблере, скорректированная для вызова из C++

```
.386
.model flat
  public _AddInts@8
.data
.code
_AddInts@8 proc
  push     EBP
  mov      EBP, ESP
  mov      EAX, DWORD PTR [EBP+8]
  add      EAX, DWORD PTR [EBP+12]
  pop      EBP
  ret      8
_AddInts@8 endp
end
```

Как видно из исходного текста, единственный параметр, который подвергся изменению по сравнению с Delphi, — это имя процедуры. Компиляция выполняется так же, как и в предыдущем варианте. Для компилятора TASM:

```
tasm32 /ml AddInts.asm
```

или для MASM:

```
ml /c AddInts.asm
```

Теперь посмотрим, как выглядит программный код, вызывающий ассемблерные процедуры из C++ .NET. Прежде всего мы должны описать вызываемую процедуру в разделе описаний переменных и функций:

```
extern "C" int _stdcall AddInts(int i1, int i2);
```

Фрагмент программы, выполняющей вычисления с использованием внешней процедуры `AddInts`, мог бы выглядеть так:

```
...
int I1 = 74;
int I2 = -56;
int ires;
...
ires = AddInts(I1, I2);
...
```

Спецификатор `"C"` запрещает компилятору C++ *декорировать* имя внешнего идентификатора. Декорирование имен (name decoration) — это стандартная технология компилятора C++, при которой происходит расширение имени с помощью дополнительных символов, несущих информацию о типе каждого параметра. Директива `extern`, как и в случае программы на Delphi, указывает на то, что идентификатор процедуры является внешним. Перед компиляцией программы на C++ необходимо добавить в проект объектный файл с вызываемой процедурой. Лучше всего, если вы скопируете объектный файл с процедурой в рабочий каталог проекта. Это замечание касается как C++ .NET, так и Delphi 7.

И еще одно замечание. Компилятор Visual C++ работает с объектными файлами в формате COFF (Common Object File Format — общий формат объектных файлов), в отличие от Delphi, который использует файлы в стандарте OMF (Object Module Format — формат объектных модулей). Поэтому в процессе сборки вашего проекта в C++ .NET вы можете получить предупреждение компоновщика:

```
Warning: converting object format from OMF to COFF
```

В принципе, это не так важно, поскольку компилятор C++ преобразует OMF-файл в COFF в любом случае. Компилятор TASM, к сожалению, не позволяет получать файлы формата COFF, а для MASM вы можете задать опцию `/coff`:

```
ml /c /coff AddInts.asm
```

Сейчас мы рассмотрим, как работает наш интерфейс, если использовать передачу параметров `cdecl` (листинг 3.16). Отличие этого метода передачи параметров от `stdcall` в том, что вызывающая процедура должна сама восстанавливать стек. Параметры передаются справа налево, как и в `stdcall`.

Листинг 3.16. Процедура на ассемблере с передачей параметров cdecl

```
.386
.model flat
public AddInts

.data
.code
AddInts proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, DWORD PTR [EBP+8]
  add     EAX, DWORD PTR [EBP+12]
  pop     EBP
  ret
AddInts endp
end
```

Команда `ret` выхода из процедуры используется здесь без параметров.

Посмотрим, как изменится наша основная программа на Delphi. Фрагмент кода, который претерпел изменения по сравнению с предыдущим примером:

```
implementation
  {$R *.dfm}
  {$L ADDINTS.OBJ}
function AddInts(X1: Integer; X2: Integer): Integer; cdecl; external;
```

В остальном исходный текст Delphi-приложения не изменился. Для корректной работы ассемблерного модуля в C++ мы опять должны внести изменения в имя процедуры (листинг 3.17). Для работы с директивой `cdecl` необходимо добавить символ подчеркивания в начало имени. Команда `ret` в процедуре вызывается без параметров.

Листинг 3.17. Ассемблерная процедура с передачей параметров cdecl для использования в C++

```
.386
.model flat
  public _AddInts
.data
```

```
.code
_AddInts proc
  push   EBP
  mov    EBP, ESP
  mov    EAX, DWORD PTR [EBP+8]
  add    EAX, DWORD PTR [EBP+12]
  pop    EBP
  ret
_AddInts endp
end
```

Что касается исходного текста программы на Visual C++, то здесь, как и в Delphi-приложении, изменения минимальны и касаются лишь раздела *деклараций*, где мы должны поменять директиву `stdcall` на `cdecl`:

```
extern "C" int _cdecl AddInts(int i1, int i2);
```

И, наконец, рассмотрим довольно широко применяемый регистровый метод передачи параметров в вызываемую функцию, представленный директивой `register` (`fastcall` в C++). Аргументы передаются через регистры `EAX`, `EDX` и `ECX` слева направо. Если имеется больше 3-х аргументов, то, начиная с 4-го, остальные передаются через стек. Посмотрим, как изменится наша процедура `AddInts` при использовании такого метода передачи аргументов (листинг 3.18).

Листинг 3.18. Ассемблерная процедура с передачей параметров `register`

```
.386
.model flat
  public AddInts
.data
.code
AddInts proc
  sub    EAX, EDX
  ret
AddInts endp
end
```

Как видим, исходный текст процедуры упростился по сравнению с другими вариантами. Этот метод действительно ускоряет работу процедур, т. к. не требуется инициализация стека и его восстановление, как при других мето-

дах передачи параметров. Однако злоупотреблять им не стоит, потому что интенсивное использование регистров процессора в процедурах будет препятствовать оптимизации программы компилятором. Как известно, для оптимизации программ многие компиляторы языков высокого уровня используют регистры процессора.

Для работы Delphi-приложения с процедурой `AddInts` необходимо в определении внешней функции указать параметр `register`:

```
implementation
  {$R *.dfm}
  {$L ADDINTS.OBJ}
function AddInts(X1: Integer; X2: Integer): Integer; register; external;
```

В Microsoft Visual C++ аналогом соглашения `register` является `fastcall`. Первые два параметра передаются через регистры `ECX` и `EDX`, остальные аргументы передаются справа налево через стек. Например, исходный текст программы на ассемблере, выполняющей операцию `I1 + I2 - I3 - I4`, где `I1`, `I2`, `I3` и `I4` — целые числа, представлен в листинге 3.19.

Листинг 3.19. Ассемблерная процедура с передачей параметров `fastcall` для использования в C++

```
.386
.model flat
  public @AddIntsR@16
.data
.code
@AddIntsR@16 proc
  push    EBP
  mov     EBP, ESP
  add     ECX, EDX
  mov     EAX, ECX
  sub     EAX, DWORD PTR [EBP+12]
  sub     EAX, DWORD PTR [EBP+8]
  pop     EBP
  ret     8
@AddIntsR@16 endp
end
```

Обратите внимание на суффикс @16 в идентификаторе имени процедуры. Он указывает на общее количество байт, занимаемых параметрами (два двойных слова в регистрах ECX и EDX и два двойных слова в стеке).

В основной программе на Visual C++ для вызова этой процедуры необходимо указать директиву fastcall:

```
extern "C" int _fastcall AddInts(int i1, int i2, int i3, int i4);
```

Автор надеется, что читатель на этих примерах понял, как разрабатывается интерфейс процедур на ассемблере с языками высокого уровня. На этом этапе мы не работали с программами, передающими в качестве параметров адреса переменных. Автор сознательно отнес эту тему к концу главы, где будут рассматриваться более сложные примеры задач.

3.3. Использование процедур на ассемблере в языках высокого уровня

Использование языка ассемблера для написания больших серьезных программ — идея утопическая, если только вы не фанат этого языка и не располагаете достаточным количеством свободного времени. Однако улучшение показателей производительности программ на языках высокого уровня невозможно без привлечения средств ассемблера. При этом улучшается производительность программы и уменьшается размер программного кода. В этой главе внимание акцентируется на совместном использовании ассемблера и программ на Delphi 7 и Visual C++ .NET. Будут рассмотрены вопросы разработки компиляции и сборки ассемблерных модулей с программами на языках высокого уровня.

Языки высокого уровня обладают мощными средствами для написания ассемблерных модулей прямо в исходном тексте программ при помощи встроенного ассемблера. Встроенный ассемблер языков высокого уровня мы рассмотрим подробнее в *главе 6*. Сейчас нас будет интересовать только разработка отдельных модулей средствами автономных компиляторов ассемблера и объединение (linking) таких модулей с программами на языках высокого уровня. Рассмотрим вначале, каким образом такие программы могут вызвать внешние процедуры.

Есть несколько вариантов объединения внешних модулей с языками высокого уровня. Первый вариант — использование объектных модулей. Ассемблер позволяет создавать объектные модули на этапе компиляции. Например, если у нас есть файл с исходным текстом на ассемблере myprog.asm, то после компиляции этого файла из командной строки:

```
tasm32 /ml myprog.asm
```

мы получим объектный модуль myprog.obj. Файл с расширением OBJ (объектный модуль) можно использовать для объединения с основной программой на языке высокого уровня. В процессе объединения объектный модуль становится частью программы.

Второй вариант — создание из нашего объектного модуля библиотеки статической или динамической компоновки. Библиотеки представляют собой определенным образом скомпонованные модули, в которые могут входить один или несколько модулей. Особенно удобны библиотеки динамической компоновки DLL (dynamic-link library), позволяющие повторно использовать процедуры и экономящие размер программы. Использование библиотек будет рассмотрено подробно в *главе 5*.

А сейчас рассмотрим использование объектных модулей. В качестве примера возьмем какую-либо процедуру или функцию на ассемблере, которую можно было бы с пользой применить в программе, написанной на Delphi или C++.

Вначале немного теории. Хочу напомнить, что мы работаем только с 32-разрядными приложениями. Такие приложения используют модель памяти `flat` и работают только с 32-разрядными операционными системами Windows 98/Windows NT/Windows 2000/Windows XP. Для такой модели не существует отдельных сегментов данных и кода. Пространство адресов считается линейным, и в нем располагаются код и данные, которые используют 32-разрядные смещения, а вызовы процедур и функций считаются ближними. Такой режим работы обеспечивает высокую производительность 32-разрядных приложений, т. к. нет преобразования "сегмент-смещение" в абсолютные адреса. Кроме того, 32-разрядные приложения "не видят" друг друга и выполняются в изолированном от других приложений пространстве адресов. Это очень сильно отличает их от 16-разрядных приложений, где все программы могли "видеть" друга.

Итак, основные принципы работы ассемблерных процедур в программах на языках высокого уровня мы разобрали. Далее, следуя принципу "лучший критерий истины — практика", перейдем к рассмотрению примеров программного кода. Мы будем рассматривать примеры вначале на Delphi, затем на Visual C++. Для совместной работы наших ассемблерных модулей и программ на языках высокого уровня будем использовать соглашение о передаче параметров `stdcall`.

Рассмотрим следующий пример. Пусть требуется найти разность двух целых чисел и вывести результат в окно приложения. Вычисление разности суммы чисел выполним в ассемблерной процедуре, скомпилированной как отдельный модуль. Результат вычитания выведем в окно основного приложения. Назовем нашу процедуру на ассемблере `subtwo`.

Исходный текст процедуры `subtwo` приведен в листинге 3.20.

Листинг 3.20. Ассемблерная процедура subtwo, вычисляющая разность двух целых чисел, для использования в Delphi

```
.386
.model flat, stdcall
  public subtwo
.data
.code
subtwo proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, DWORD PTR [EBP+8]
  sub     EAX, DWORD PTR [EBP+12]
  pop     EBP
  ret     8
subtwo  endp
end
```

Сохраним исходный текст процедуры в файле subtwo.asm и откомпилируем программу при помощи одной из двух командных строк в зависимости от вида компилятора ассемблера:

```
tasm32 /ml subtwo.asm subtwo.obj
ml /c /Fo subtwo.obj subtwo.asm
```

Если в тексте программы нет ошибок, мы получим объектный файл subtwo.obj, готовый к употреблению.

Для передачи параметров в процедуру `subwo` используется стек. Для доступа к параметрам мы используем регистр `EBP`, текущее значение которого процедура сохраняет в стеке. Помним, что мы имеем дело с 32-разрядными операндами, поэтому первый параметр `I1` будет иметь смещение в стеке на `8` (с учетом помещенного в стек `EBP`), а второй параметр `I2` — смещение `12` относительно вершины стека (см. рис 3.1).

Расположение параметров в стеке соответствует соглашению для директивы `stdcall`. Сама процедура вычитает числа и возвращает результат в регистре `EAX`. Согласно `stdcall` вызываемая процедура должна сама восстанавливать (очищать) стек. Перед последней командой `ret` возвращаемое значение уже находится в регистре `EAX`, а в стеке все еще присутствуют два параметра. Чтобы восстановить стек, необходимо удалить параметры из

Хочется сделать несколько общих замечаний по работе с внешними процедурами и функциями. Они касаются не только процедуры subtwo.

1. Желательно сохранять в стеке регистры EBX, EBP, ESI и EDI, т. к. они могут использоваться операционной системой, и разрушение их содержимого может привести к неприятным последствиям. Регистры EAX, EDX и ECX можно использовать по своему усмотрению, не заботясь об их сохранении.
2. В 32-разрядных приложениях понятие "сегментный регистр" отсутствует. Если пытаться по аналогии с MS-DOS использовать регистры CS, ES и DS, то это немедленно приведет к краху программы.
3. Не забывайте освобождать стек при выходе из процедуры при использовании директив pascal, stdcall и safecall. Количество удаляемых байт равно количеству параметров, умноженному на 4. К примеру, для удаления из стека 3-х параметров необходимо в процедуре последней вызвать команду ret 12.

   Вместо ret n можно использовать комбинацию команд:

```
add     ESP, n
ret
```

Вернемся к нашему примеру. Разработаем программу на языке высокого уровня, вызывающую нашу процедуру subtwo. Начнем с Delphi 7. Наша программа будет представлять собой обычное оконное приложение Windows. Разместим на главной форме приложения три поля редактирования Edit и назовем их I1Edit, I2Edit и subResult. Поле редактирования I1Edit будем использовать для ввода числа I1, поле I2Edit — для ввода I2 и поле subResult — для отображения результата вычитания I2 из I1. Слева от полей редактирования поместим три метки статического текста Label.

Разместим на форме также кнопку Button. При нажатии этой кнопки в работающем приложении в поле редактирования subResult будет отображаться результат вычитания.

Теперь внесем изменения в исходный текст программы. В секцию implementation добавим описание нашей процедуры subtwo:

```
implementation
  {$R *.dfm}
  {$L subtwo.obj}
```

Последняя строка сообщает компоновщику, какой внешний модуль будет использоваться в основной программе. В данном случае это файл subtwo.obj.

Должен заметить, что имя файла объектного модуля может и не совпадать с именем вызываемой процедуры или функции. Необязательно также, чтобы модуль subtwo.obj содержал процедуру subtwo с тем же именем. Более того, объектный модуль может содержать несколько процедур или функций.

Следующей строкой можно объявить вызываемую процедуру:

```
function subtwo(i1: Integer; i2: Integer): Integer; stdcall; external;
```

Директива external указывает на то, что вызываемая процедура является внешней по отношению к вызывающей программе и находится в другом, возможно внешнем модуле. Директива stdcall указывает компилятору правила, по которым параметры i1 и i2 обрабатываются в вызываемой процедуре.

Далее напишем обработчик нажатия кнопки (листинг 3.21).

Листинг 3.21. Обработчик нажатия кнопки

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I1, I2: Integer;
begin
  I1 := StrToInt(I1Edit.Text);
  I2 := StrToInt(I2Edit.Text);
  subResult.Text := IntToStr(subtwo(I1, I2));
end;
```

Обработчик представляет собой обычную процедуру, в которой мы объявили две переменные целого типа I1 и I2 и написали несколько операторов. Введенные в поля редактирования текстовые представления I1 и I2 преобразуются в целые числа при помощи функции StrToInt:

```
I1 := StrToInt(I1Edit.Text);
I2 := StrToInt(I2Edit.Text);
```

Далее программа отображает результат вычитания в поле редактирования subResult при помощи оператора:

```
subResult.Text := IntToStr((subtwo(I1, I2)));
```

Здесь целочисленный результат, возвращаемый процедурой subtwo, преобразуется в текст функцией IntToStr и отображается в поле редактирования.

Как вы заметили, процедура `subtwo` использует в качестве параметров двойные слова (`DWORD`) и возвращает результат также в виде двойного слова. Компилятор Delphi корректно преобразует целочисленные величины типа `Integer` в двойные слова, поэтому ошибок не возникает. Мы могли бы, к примеру, в секции определения переменных нашей программы написать:

```
var
  I1, I2: DWORD;
```

Функции `StrToInt` и `IntToStr` в этом случае отрабатывают корректно. Нужно быть очень внимательным при преобразовании типов, т. к. ошибки преобразования очень трудно поддаются анализу и не всегда отслеживаются компиляторами! Это касается как приложений на Delphi, так и на Visual C++ .NET.

Вид окна работающего приложения представлен на рис. 3.2.

Рис. 3.2. Окно приложения, выполняющего вычитание двух целых чисел

Как видно из рисунка, работающее приложение ожидает ввода значений в полях `Integer 1` и `Integer 2` и при нажатии на кнопку `Get I1-I2` отображает в поле редактирования `Sub I1-I2` результат вычитания.

И еще одно. Не забудьте скопировать файл subtwo.obj в рабочий каталог нашего проекта, иначе компоновщик не сможет найти объектный файл с нашей процедурой.

Рассмотрим этот же пример, но в реализации Microsoft Visual C++ .NET (листинг 3.22). Первое, что мы сделаем — это изменим идентификаторы имен в нашей ассемблерной процедуре.

Листинг 3.22. Ассемблерная процедура, вычисляющая разность двух целых чисел, для использования в C++

```
.386
.model flat
  public _subtwo@8
.data
```

```
.code
_subtwo@8 proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, DWORD PTR [EBP+8]
  sub     EAX, DWORD PTR [EBP+12]
  pop     EBP
  ret     8
_subtwo@8 endp
end
```

Имя процедуры subtwo изменилось на _subtwo@8 в соответствии с соглашением stdcall для работы с компилятором C++ .NET. Скомпилируем наш ассемблерный модуль:

```
ml /c /Fo subtwo.obj subtwo.asm
```

Теперь можно разработать приложение, использующее процедуру subtwo. Воспользуемся для этого Мастером приложений C++ .NET. Создадим MFC-приложение (Microsoft Foundation Classes — библиотека базовых классов Microsoft) на основе диалогового окна. Позволим мастеру сконструировать каркас приложения, после чего внесем некоторые изменения в проект.

Удалим все элементы управления с главной формы приложения. Затем поместим на нашу форму, как и в приложении на Delphi, три элемента статического текста Static Text, три поля редактирования Edit и кнопку Button.

В нашем приложении придется оперировать с элементами управления как с переменными. Для этого необходимо поставить в соответствие элементу управления переменную того или иного типа. Например, если мы решили, что элемент управления IDC_EDIT1 (первое поле редактирования Edit) будет принимать в качестве ввода значение переменной целого типа I1, то можно связать элемент и переменную. В этом случае все программные манипуляции с элементом управления будут сказываться на значении соответствующей переменной и наоборот. Для синхронизации всех таких изменений служит функция UpdateData. По аналогии свяжем элемент управления IDC_EDIT2 (второе поле Edit) с целочисленной переменной I2, а элемент управления IDC_EDIT3 (третье поле Edit) — с целочисленной переменной iSubResult. В классе диалогового окна определим внешнюю функцию subtwo:

```
extern "C" int _stdcall subtwo(int i1, int i2);
```

Спецификатор "C" защищает оригинальное имя функции от декорирования (изменения). Если не указывать этот спецификатор, то компилятор

Visual C++ .NET изменит оригинальное имя функции при помощи дополнительных символов, несущих информацию о типе каждого параметра. Эта информация используется компоновщиком при создании исполняемого файла. Далее напишем обработчик события при нажатии кнопки. Соответствующий фрагмент кода приведен в листинге 3.23.

Листинг 3.23. Обработчик события при нажатии кнопки в программе на С++

```
void CSUBSTRACTIONTWOINTSINCNETDlg::OnBnClickedButton1()
{
  UpdateData(TRUE);
  iSubResult = subtwo(I1, I2);
  UpdateData(FALSE);
}
```

Функция `UpdateData` с параметром `TRUE` в обработчике нажатия кнопки позволяет обновить значения переменных, соответствующих элементам управления. Иными словами, эта функция передает текущее содержимое элементов управления на экране в переменные, связанные с этими элементами управления.

Далее, в переменную `iSubResult` передается результат выполнения процедуры `subtwo`. И, наконец, мы обновляем содержимое элементов управления в соответствии со связанными с ними переменными. Это действие выполняет функция `UpdateData` с аргументом `FALSE`.

После компиляции программы можно выполнить наше приложение. Окно работающего приложения изображено на рис. 3.3.

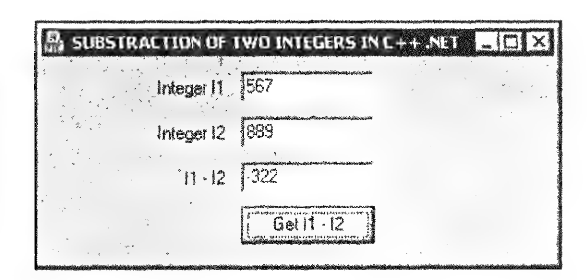


Рис. 3.3. Окно приложения, выполняющего вычитание двух целых чисел

Рассмотрим предыдущий пример, но с другими условиями передачи параметров. В качестве параметров, передаваемых в вызываемую процедуру на ассемблере, мы будем использовать не значения переменных, а их адреса. Для операций с переменными, представленными адресами в памяти, используются указатели. Указатель — это переменная, содержащая адрес, по

которому находится в памяти другая переменная. Указатели очень широко применяются в языках высокого уровня. Например, работа со строками данных и массивами в основном выполняется с использованием указателей. Передача массивов и строк в качестве параметров в подпрограммы также осуществляется при помощи указателей. Во многих случаях программисты используют подпрограммы, возвращающие в качестве результата указатель, т. е. адрес переменной.

Применение указателей значительно расширяет возможности обработки данных в программе. Далее мы часто будем иметь дело с указателями, сейчас же рассмотрим пример, где они применяются. Для начала изменим исходный текст ассемблерной процедуры. Назовем нашу процедуру `subtwop`. Исходный текст представлен в листинге 3.24.

Листинг 3.24. Ассемблерная процедура с использованием указателей в качестве параметров

```
.386
.model flat
  public subtwop
.data
  subRes  DD 0
.code
subtwop proc
  push    ESI
  push    EBP
  mov     EBP, ESP
  mov     ESI, DWORD PTR [EBP+12]
  mov     EAX, [ESI]
  mov     ESI, DWORD PTR [EBP+16]
  sub     EAX, [ESI]
  mov     subRes, EAX
  mov     EAX, offset subRes
  pop     EBP
  pop     ESI
  ret     8
subtwop  endp
end
```

Как видите, в исходном тексте произошли существенные изменения. Для извлечения адреса переменной из стека и передачи значения, хранящегося

по этому адресу, используется регистр `ESI`. Переменная `I1` помещается в регистр `EAX` с помощью двух команд:

```
mov     ESI, DWORD PTR [EBP+12]
mov     EAX, [ESI]
```

Поскольку мы сохранили предварительно содержимое регистра `ESI` в стеке, то переменная `I1` хранится по адресу `EBP + 12`, а переменная `I2` — по адресу `EBP + 16`.

Команды:

```
mov     subRes, EAX
mov     EAX, offset subRes
```

сохраняют результат в переменной `subRes`, а в регистре `EAX` возвращают адрес этой переменной. Посмотрим, как изменится исходный текст приложения на Delphi по сравнению с предыдущим примером. В секции `implementation` переопределим нашу процедуру:

```
implementation
  {$R *.dfm}
  {$L subtwop.obj}
function subtwop(pi1:PInteger; pi2:Pinteger):PInteger; stdcall; external;
```

В обработчике нажатия кнопки исходный текст также изменится (листинг 3.25).

Листинг 3.25. Обработчик нажатия кнопки в программе на Delphi 7

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I1, I2: Integer;
begin
  I1 := StrToInt(I1Edit.Text);
  I2 := StrToInt(I2Edit.Text);
  subResult.Text := IntToStr(subtwop(@I1, @I2)^);
end;
```

В модифицированном варианте программы интерес вызывает строка

```
subResult.Text:= IntToStr (subtwop (@I1, @I2)^);
```

Символы "@" перед переменными I1 и I2 означают операцию взятия адреса этих переменных, т. к. процедура subtwop в качестве параметров принимает указатели типа PInteger. Символ "^" после идентификатора процедуры означает *разыменование* указателя (dereferencing). Так как subtwop возвращает указатель на переменную целого типа, то разыменование возвращает целое значение, находящееся по этому адресу. Функция IntToStr принимает в качестве параметра целочисленную переменную, поэтому все наши преобразования корректны.

Использование указателей в Delphi — это отдельная тема, но она настолько важна, что необходимо остановиться на ней более подробно. Применение указателей лучше всего продемонстрировать на примере (листинг 3.26).

Листинг 3.26. Пример использования указателей в программе на Delphi

```
var
  I1, I2: Integer;
  P: ^Integer;
begin
  I1 := 10;
  P := @I1;
  I2 := P^;
end;
```

Переменные I1 и I2 объявляются как целые, а переменная P — как указатель на целое. Затем переменной I1 присваивается значение 10. Строка:

```
P := @I1;
```

присваивает указателю P адрес, где хранится переменная I1. Оператор:

```
I2 := P^;
```

выполняет разыменование указателя P и присваивает переменной I2 значение, находящееся по адресу P. Таким образом, в результате выполнения этого фрагмента программного кода переменной I2 будет присвоено значения I1, т. е. I2 = 10. Оператор взятия адреса "@" применяется также и для работы с процедурами и функциями.

Программный код ассемблерной процедуры для работы с приложением на C++ .NET изменится в части именования функции subtwop (листинг 3.27).

Листинг 3.27. Ассемблерная процедура для работы с программой на C++

```
.386
.model flat
  public _subtwop@8
.data
  subRes DD 0
.code
_subtwop@8 proc
   ...
_subtwop@8 endp
end
```

Фрагмент кода для вызова ассемблерной процедуры из C++ .NET приведен в листинге 3.28.

Листинг 3.28. Фрагмент кода для вызова ассемблерной процедуры из программы на C++

```
void CSubstractTwoByPointersDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here
  UpdateData(TRUE);
  SubRes = *subtwop(&I1, &I2);
  UpdateData(FALSE);
}
```

Здесь переменная `SubRes` связана с полем редактирования `Edit`, в котором отображается результат вычитания, а `I1` и `I2` связаны с полями редактирования для ввода целых чисел.

Поскольку наша процедура принимает в качестве параметров указатели и возвращает указатель, то объявление внешней процедуры `subtwop` в классе диалогового окна несколько изменится:

```
extern "C" int* _stdcall subtwop(int *pi1, int *pi2);
```

В основной программе наибольший интерес вызывает строчка:

```
SubRes = *subtwop(&I1, &I2);
```

6 Зак. 1064

Здесь, так же как и в Delphi, мы имеем дело с операциями взятия адреса и разыменованием указателей. В C++ выражения типа &I1 и &I2 возвращают адреса переменных I1 и I2. Напомню, что эти переменные связаны с соответствующими элементами управления. Символ "*" перед именем процедуры subtwop является оператором разыменования. Переменной SubRes присваивается значение переменной по адресу, который возвращает процедура subtwop. Читатель может самостоятельно проверить результат, скомпилировав приложение.

Использование указателей очень эффективно при обработке строк и массивов данных. Обычно ассемблерные процедуры для доступа к элементам таких массивов используют регистры ESI и EDI, хотя можно применять для индексации и другие регистры. Следующие несколько примеров демонстрируют работу с массивами данных с использованием указателей.

В следующем примере необходимо найти максимальное значение в массиве целых чисел. Вычисление максимального элемента массива будет выполнять ассемблерная процедура.

Исходный текст процедуры на ассемблере (назовем ее maxint) приведен в листинге 3.29.

Листинг 3.29. Ассемблерная процедура maxint

```
.386
.model flat
  public maxint
.data
.code
maxint proc
  push    ESI
  push    EBP
  mov     EBP, ESP
  mov     EDX, DWORD PTR [EBP+16]
  mov     ESI, DWORD PTR [EBP+12]
next_cmp:
  mov     EAX, DWORD PTR [ESI]
  cmp     EAX, DWORD PTR [ESI+4]
  jle     dec_cnt
  xchg    EAX, DWORD PTR [ESI+4]
dec_cnt:
  dec     EDX
  cmp     EDX, 0
```

```
  jz      fin
  add     ESI, 4
  jmp     next_cmp
fin:
  mov     EAX, ESI
  pop     EBP
  pop     ESI
  ret     8
maxint endp
end
```

Процедура maxint принимает два параметра: адрес массива (EBP + 12) и размер массива (EBP + 16). В начале цикла в регистр EAX помещается первый элемент массива и сравнивается с последующим элементом. Если содержимое регистра EAX больше содержимого ячейки памяти, то происходит обмен между регистром и памятью.

В следующей итерации в регистр EAX загружается большее из двух чисел предыдущей итерации и снова происходит сравнение. К концу цикла процедура определит адрес, по которому находится максимальный элемент массива. Этот адрес будет возвращен в основную программу в регистре EAX. Счетчиком итераций является регистр EDX, принимающий в качестве параметра размер массива. В каждой итерации адрес элемента массива, содержащийся в регистре ESI, увеличивается на 4, т. к. целое число занимает 4 байта.

Приложение, вызывающее процедуру maxint, разработаем вначале в среде Delphi 7. На главной форме приложения разместим визуальные компоненты: два поля редактирования Edit, две метки статического текста Label и кнопку Button. Напишем два обработчика событий — нажатие кнопки и инициализация в момент, когда главное окно приложения становится активным.

В секции var исходного модуля определим переменные, с которыми будем работать. Зададим массив I1 из 12-ти целых чисел, его размер как целое число SI1, целочисленную переменную Max, в которую поместим результат выполнения процедуры maxint, и, наконец, указатель PMax целочисленного типа.

Кроме того, не забудем поместить в секцию implementation описание нашей внешней процедуры. Исходный текст модуля приложения показан в листинге 3.30, он содержит все обработчики и декларации, о которых только что было упомянуто.

Листинг 3.30. Программа на Delphi, вызывающая ассемблерную процедуру `maxint`

```
uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1:  TButton;
    Edit1:    TEdit;
    Edit2:    TEdit;
    Label1:   TLabel;
    Label2:   TLabel;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);

    private
     { Private declarations }
    public
     { Public declarations }
end;

var
  Form1: TForm1;
  I1: array [1..12] of Integer = (585, 1751, -27, -76, 312, 93,
                                  5, -1, 57, 22, -5, 997);
  SI1: Integer;
  pMax: PInteger;
  Max: Integer;

implementation

  {$R *.dfm}
  {$L  maxint.obj}

function maxint(PI1: PInteger; SI1: Integer):PInteger; stdcall; external;
```

```
// Обработчик нажатия кнопки
procedure TForm1.Button1Click(Sender: TObject);
begin
  pMax := maxint(Addr(I1), SI1);
  Max := pMax^;
  Edit2.Text := IntToStr(Max);
end;

// Действия по инициализации приложения
procedure TForm1.FormCreate(Sender: TObject);
var
  Cnt: Integer;
begin
  SI1 := SizeOf(I1) div 4;
  for Cnt := 1 to SI1 do
    Edit1.Text := Edit1.Text + '  ' + IntToStr(I1[Cnt]);
end;
end.
```

Еще несколько слов об обработчиках событий. Обработчик `FormCreate` записывает в переменную `SI1` размер массива `I1`. Поскольку размер массива возвращается в байтах, необходимо привести это значение к размерности двойного слова, для чего следует разделить полученный размер массива на 4. Именно это и сделано в операторе:

```
SI1 := SizeOf(I1) div 4;
```

Далее в цикле `for` происходит вывод значений элементов массива в поле редактирования `Edit1`.

Обработчик нажатия кнопки присваивает указателю `pMax` результат, возвращаемый процедурой `maxint`. Обратите внимание, что первым аргументом является адрес массива `I1`. Для получения адреса массива используется оператор взятия адреса `Addr`. Этот оператор является аналогом оператора `"@"`, поэтому вместо выражения:

```
pMax := maxint(Addr(I1), SI1);
```

можно написать:

```
pMax := maxint(@I1, SI1);
```

Далее выполняется операция разыменования указателя, и в переменную Max заносится значение переменной, находящейся по адресу pMax.

И последнее, что остается сделать в обработчике кнопки — вывести результат в поле редактирования Edit2:

```
Edit2.Text := IntToStr(Max);
```

Окно работающего Delphi-приложения изображено на рис. 3.4.

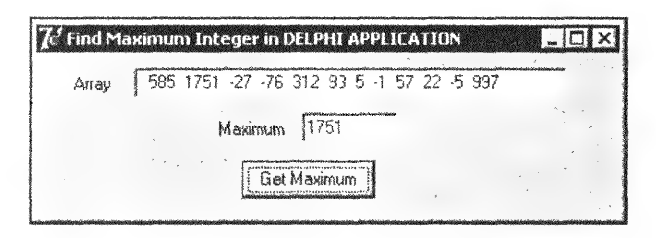


Рис. 3.4. Окно приложения, выполняющего поиск максимума в массиве целых чисел

Изменим вариант решения задачи для Visual C++ .NET. Наверное, будет полезно увидеть другое исполнение программы с другим интерфейсом. Ассемблерный вариант процедуры (назовем ее maxval) для этого случая (листинг 3.31) будет выглядеть иначе, чем в примере для Delphi.

Листинг 3.31. Ассемблерная процедура maxval, используемая в программе на C++

```
.386
.model flat
  public _maxval@8
.data
  MaxVal  DD 0
.code
_maxval@8 proc
  push    ESI
  push    EBP
  mov     EBP, ESP
  mov     ECX, DWORD PTR [EBP+16]
  dec     ECX
  mov     ESI, DWORD PTR [EBP+12]
  mov     EAX, [ESI]
```

```
next_cmp:
  cmp     EAX, [ESI+4]
  jg      go_loop
  mov     EAX, [ESI+4]
go_loop:
  add     ESI, 4
  loop    next_cmp
  mov     DWORD PTR MaxVal, EAX
  mov     EAX, offset MaxVal
  pop     EBP
  pop     ESI
  ret     8
_maxval@8 endp
end
```

Для хранения максимального элемента массива здесь используется переменная MaxVal, а результатом выполнения процедуры является адрес этой переменной, который возвращается в вызывающую программу в регистре EAX. Поскольку мы имеем дело с компилятором C++, то необходимо соответствующим образом определить имя процедуры maxval. В соответствии с соглашением stdcall она примет вид _maxval@8. Напомню командную строку для макроассемблера MASM:

```
ml /c maxval.asm
```

Основную программу в C++ .NET построим на основе класса диалогового окна, причем не будем использовать в качестве элементов управления ни кнопки, ни поля редактирования. Единственное, что мы сделаем — это разместим на форме приложения два элемента статического текста. Для вывода результатов работы приложения будем использовать клиентскую область окна приложения. При нажатии левой кнопки мыши (событие WM_LBUTTONDOWN) в окне приложения будет отображаться как массив целых чисел, так и максимальный элемент этого массива.

Обработчик события WM_LBUTTONDOWN представлен в листинге 3.32.

Листинг 3.32. Обработчик нажатия левой кнопки мыши в программе на C++

```
void CFindMaxIntegerinArrayDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
  // TODO: Add your message hand'er code here and/or call default

  int  i1[10] = {4, 87, 90, -34, 2, -5696, -45, 76, -12, -964};
```

```
  char buf[8];
  CClientDC dc(this);
  RECT rect;
  GetClientRect(&rect);
  for (int cnt = 0; cnt < sizeof(il)/4; cnt++)
  {
    dc.TextOut((rect.right - rect.left)/30 + cnt*45,
               (rect.bottom - rect.top)/4,
                itoa(il[cnt], buf, 10));
  };
  int ires = *maxval(il, sizeof(il)/4);
  dc.TextOut((rect.right - rect.left)/2 - 40,
             (rect.bottom - rect.top)/2 + 30,
              itoa(ires, buf,10));

  CDialog::OnLButtonDown(nFlags, point);
}
```

Как обычно, объявим нашу процедуру `maxval` в разделе деклараций:

```
extern "C" int* _stdcall maxval(int *pil, int sil);
```

В качестве параметров она принимает адрес и размер массива, а возвращает адрес, по которому будет помещен максимальный элемент массива.

Программный код обработчика нажатия левой кнопки мыши предназначен для отображения наших данных в клиентской области окна. Для отображения текста или графики в окне приложения необходимо вначале получить *контекст устройства* отображения. Он представляет собой структуру данных Windows, в которой содержится информация об атрибутах рисования для таких устройств, как дисплей или принтер. Все запросы на вывод графики и текста проходят через объект контекста устройства, который включает в себя все WIN API функции для прорисовки линий, фигур и текста.

Контекст устройства позволяет выполнить аппаратно независимую процедуру рисования в Windows, он может использоваться для вывода графической информации на экран, принтер или в метафайл. Объект `CClientDC` инкапсулирует основные функции контекста устройства для работы с клиентской областью окна. Функция `GetClientRect` определяет координаты клиентской части окна приложения и сохраняет их в структуре `RECT`. Привязка координат рисования выполняется по отношению к левому верхнему углу клиентской области, координаты которого равны (0, 0). Функция

`TextOut` записывает строку, начиная с координат, указанных первыми двумя параметрами. После такого теоретического отступления анализ программного кода в обработчике нажатия кнопки значительно упростится.

Для получения атрибутов контекста устройства и подготовки рисования в клиентской области окна используются следующие операторы:

```
CClientDC dc(this);
RECT rect;
GetClientRect(&rect)
```

Для вывода в клиентскую область окна текстового представления массива целых чисел применяется фрагмент кода:

```
for (int cnt = 0; cnt < sizeof(i1)/4; cnt++)
{
  dc.TextOut((rect.right - rect.left)/30 + cnt*45,
             (rect.bottom - rect.top)/4,
              itoa(i1[cnt], buf, 10));
};
```

Преобразование элемента массива в строку выполняет функция:

```
itoa(i1[cnt], buf, 10)
```

Результат выполнения процедуры `maxval` сохраняется в переменной `ires` для дальнейшего использования. Поскольку процедура возвращает указатель, то необходимо выполнить операцию разыменования при помощи оператора "`*`":

```
int ires = *maxval(i1, sizeof(i1)/4);
```

И, наконец, вывод результата поиска на экран выполняется знакомой нам уже функцией `TextOut`:

```
dc.TextOut((rect.right - rect.left)/2 - 40,
           (rect.bottom - rect.top)/2 + 30,
            itoa(ires, buf,10));
```

Окно работающего приложения изображено на рис. 3.5.

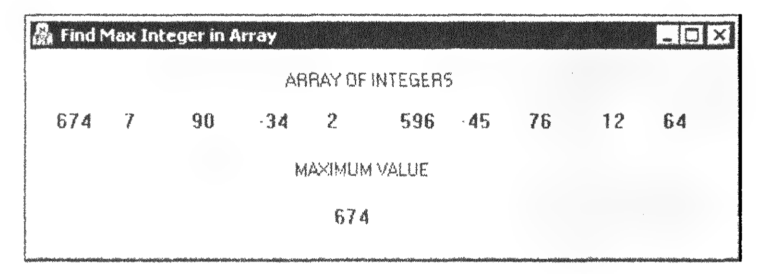


Рис. 3.5. Окно приложения, выполняющего поиск максимального элемента в массиве целых чисел

Для нахождения минимального элемента в массиве целых чисел необходимо внести небольшие изменения в нашу процедуру на ассемблере (листинг 3.33).

Листинг 3.33. Ассемблерная процедура, выполняющая поиск минимального элемента массива

```
.386
.model flat
  public _minval@8
.data
  MinVal  DD 0
.code
_minval@8 proc
  push    ESI
  push    EBP
  mov     EBP, ESP
  mov     ECX, DWORD PTR [EBP+16]
  dec     ECX

  mov     ESI, DWORD PTR [EBP+12]
  mov     EAX, [ESI]
next_cmp:
  cmp     EAX, [ESI+4]
  jle     go_loop
  mov     EAX, [ESI+4]
go_loop:
  add     ESI, 4
```

```
  loop     next_cmp
  mov      DWORD PTR MinVal, EAX
  mov      EAX, offset MinVal
  pop      EBP
  pop      ESI
  ret      8
_minval@8 endp
end
```

До сих пор мы работали с целыми числами. Следующие несколько примеров демонстрируют, как можно в процедурах на ассемблере выполнять операции с вещественными числами. Рассмотрим вначале простой пример суммирования двух вещественных чисел и вывода результата в окно приложения.

Операция суммирования выполняется в ассемблерной процедуре, возвращающей в основную программу адрес ячейки памяти, в которой находится результат суммирования.

Исходный текст ассемблерной процедуры для работы с Delphi-приложением представлен в листинге 3.34.

Листинг 3.34. Ассемблерная процедура, выполняющая суммирование двух вещественных чисел

```
.386
.model flat
  public addreals
.data
  SUM    DD 0
.code
addreals proc
  push    EBX
  push    EBP
  mov     EBP, ESP
  mov     EBX, DWORD PTR [EBP+12]
  mov     EDX, DWORD PTR [EBP+16]
  finit
  fld     DWORD PTR [EBX]
  fadd    DWORD PTR [EDX]
  fstp    DWORD PTR SUM
  fwait
```

```
  mov     EAX, offset SUM
  pop     EBP
  pop     EBX
  ret     8
addreals endp
end
```

Первое слагаемое передается в процедуру со смещением 12 в стеке, второе — со смещением 16. Операцию сложения будем выполнять с использованием функций математического сопроцессора (блока выполнения операций с плавающей запятой для процессоров Pentium). Основные команды математического сопроцессора были нами рассмотрены в *главе 2*, поэтому остановимся на них лишь вкратце.

Команда `finit` инициализирует сопроцессор. Последующие две команды выполняют операцию суммирования и временно сохраняют результат в вершине стека сопроцессора. Команда:

```
fstp    DWORD PTR SUM
```

выталкивает результат операции в ячейку памяти и освобождает вершину стека сопроцессора. Адрес ячейки памяти передается в вызывающую программу на Delphi через регистр `EAX`. Откомпилируем наш ассемблерный модуль при помощи турбо ассемблера TASM:

```
TASM32 /ml addreals.asm addreals.obj
```

Демонстрационную программу на Delphi разработаем следующим образом: разместим на главной форме приложения три поля редактирования `Edit` вместе с метками статического текста `Label` и кнопку `Button`. При нажатии на кнопку в поле редактирования с заголовком "X1 + X2" отобразится сумма чисел, введенных пользователем в полях "X1" и "X2".

Сделаем некоторые изменения в исходном тексте шаблона приложения. В секции `implementation` объявим ассемблерную процедуру:

```
implementation
  {$R *.dfm}
  {$L addreals.obj}
function addreals(pX1: PSingle;pX2: PSingle): PSingle; stdcall; external;
```

Обратите внимание на то, как описаны указатели в параметрах процедуры и возвращаемый результат. Они имеют тип PSingle и указывают на переменные вещественного типа Single.

Напишем обработчик нажатия кнопки для нашего приложения (листинг 3.35).

Листинг 3.35. Обработчик нажатия кнопки в программе на Delphi

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  X1 := StrToFloat(Edit1.Text);
  X2 := StrToFloat(Edit2.Text);
  Edit3.Text := FloatToStrF((addreals(@X1, @X2))^, ffGeneral, 5, 7);
end;
```

Этот фрагмент кода выполняет следующие действия:

- записывает в переменные X1 и X2 значения вещественных чисел из полей X1 и X2. Поскольку поле редактирования Edit интерпретирует введенные символы как текст, мы должны преобразовать этот текст в вещественное число. Сделать это довольно просто — в Delphi есть специальная функция StrToFloat, преобразующая строку в вещественное число;
- вызывает внешнюю процедуру addreals с адресами переменных X1 и X2 в качестве параметров и получает адрес суммы в качестве результата. Далее через операцию разыменования указателя результат сложения передается в функцию FloatToStrF. Эта функция преобразует вещественное число в строковую переменную, которую легко вывести в поле редактирования. Все вышеописанные действия выполняет один оператор:

```
Edit3.Text := FloatToStrF((addreals(@X1, @X2))^, ffGeneral,5,7);
```

Хочу напомнить, что оператор "@" возвращает адрес переменной, а оператор "^", стоящий после имени переменной-указателя, возвращает значение по этому адресу.

Окно работающего Delphi-приложения изображено на рис. 3.6.

Чтобы решить ту же задачу средствами Visual C++ .NET, мы должны прежде всего изменить имя процедуры в ассемблерном модуле в соответствии с требованиями компилятора C++. Напомним, что во всех наших примерах мы используем соглашение stdcall. Тогда фрагменты кода, где встречается процедура addreals, будут выглядеть следующим образом (листинг 3.36).

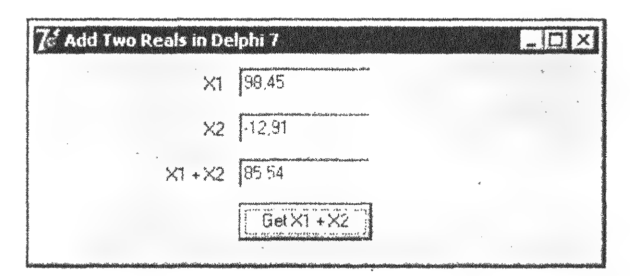


Рис. 3.6. Окно приложения, выполняющего операцию суммирования двух вещественных чисел

Листинг 3.36. Ассемблерная процедура `addreals` для использования в программе на C++

```
.386
.model flat
  public _addreals@8
.data
  SUM  DD 0
.code
_addreals@8 proc
  push    EBX
  push    EBP
   ...
  ret     8
_addreals@8 endp
end
```

За основу приложения в Visual C++ .NET возьмем диалоговое окно, на котором разместим, как и в Delphi-варианте, три поля редактирования `Edit` с тремя элементами статического текста `Static Text` и кнопку `Button`. В классе окна объявим процедуру `addreals` как внешнюю с соответствующими параметрами:

```
extern "C" float* _stdcall addreals(float *x1, float *x2);
```

Свяжем с элементами управления `Edit` три вещественные переменные `X1`, `X2` и `sumX`. Теперь напишем обработчик нажатия кнопки, исходный текст которого приведен в листинге 3.37.

Листинг 3.37. Обработчик события нажатия кнопки в программе на C++

```
void CAddTwoRealsinCNETDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  UpdateData(TRUE);
  sumX = *addreals(&X1, &X2);
  UpdateData(FALSE);
}
```

Приложение ожидает ввода переменных X1 и X2 в поля редактирования. Функция UpdateData с параметром TRUE передает текущие значения элементов управления в связанные с ними переменные X1 и X2. Процедура addreals получает в качестве параметров адреса этих переменных и возвращает адрес, по которому находится сумма. Оператор "*" позволяет передать в переменную sumX сумму двух вещественных чисел.

Последняя команда UpdateData с параметром FALSE передает текущие значения переменных в связанные с ними элементы управления и обновляет экран. Окно работающего приложения изображено на рис. 3.7.

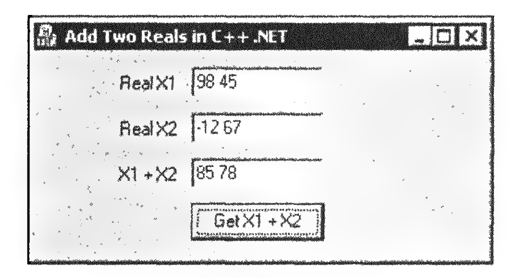


Рис. 3.7. Окно приложения, выполняющего суммирование двух вещественных чисел

Наш следующий пример более сложен и показывает, как можно найти максимальный элемент в массиве вещественных чисел. Размерность массива зададим равной 9. Разработаем классическое процедурно-ориентированное приложение в Visual C++ .NET. В таком приложении обычно присутствуют два взаимосвязанных фрагмента кода: главная процедура WinMain, регистрирующая класс окна и выполняющая все функции по инициализации экземпляра окна приложения, и функция обратного вызова (оконная процедура). Более подробно мы будем рассматривать разработку таких приложений в *главе 4*, а сейчас просто используем каркас приложения, построенный для нас Мастером приложений C++ .NET.

Поиск максимального элемента в массиве вещественных чисел выполним в ассемблерной процедуре, исходный текст которой представлен в листинге 3.38.

Листинг 3.38. Ассемблерная процедура, выполняющая поиск максимального элемента в массиве вещественных чисел

```
.386
.model flat
  public _maxreal@8
.data
  MAXREAL DD 0
.code
_maxreal@8 proc
  push    EBX
  push    EBP
  mov     EBP,ESP
  mov     EBX, DWORD PTR [EBP+12]
  mov     EDX, DWORD PTR [EBP+16]
  mov     ECX, 1
  finit
  fld     DWORD PTR [EBX]
NEXT_CMP:
  add     EBX, 4
  fcom    DWORD PTR [EBX]
  fstsw   AX
  sahf
  jnc     CHECK_INDEX
  fld     DWORD PTR [EBX]
CHECK_INDEX:
  cmp     ECX, EDX
  je      FIN
  inc     ECX
  jmp     NEXT_CMP
FIN:
  fwait
  fstp    DWORD PTR MAXREAL
  mov     EAX, offset MAXREAL
  pop     EBP
```

```
  pop     EBX
  ret     8
_maxreal@8 endp
end
```

В этой процедуре используются команды математического сопроцессора. Как обычно, для извлечения параметров используется регистр EBP. По смещению 12 в стеке находится адрес массива вещественных чисел или адрес первого элемента, что одно и то же. Размер массива находится по смещению 16 в стеке. После обработки массива адрес максимального элемента передается, как обычно, в регистре EAX. Текущее значение максимума программа сохраняет в локальной переменной MAXREAL. Мы не можем передавать ни значение переменной, ни тем более ее адрес сразу в регистре EAX. Причиной этого является тот факт, что команда математического сопроцессора fstp передает значение из стека сопроцессора только в ячейку памяти. Поэтому для возвращения результата в вызывающую программу мы используем команды:

```
fstp    DWORD PTR MAXREAL
mov     EAX, offset MAXREAL
```

Наша процедура возвращает адрес переменной MAXREAL, по которому находится максимальный элемент массива. Можно усложнить процедуру и вычислять адрес максимального элемента без использования промежуточной переменной MAXREAL. Читатели могут попробовать разработать такую процедуру самостоятельно в качестве упражнения.

Теперь разработаем программу на C++ .NET, которая будет использовать результат выполнения ассемблерного модуля. Воспользуемся Мастером приложений и разработаем обычное 32-разрядное Windows-приложение. Исходный текст процедуры WinMain и функции обратного вызова представлен в листинге 3.39.

Листинг 3.39. Программа на C++, вызывающая ассемблерную процедуру

```
#include "stdafx.h"
#include "Find Max Value in Array of Reals.h"
#define  MAX_LOADSTRING 100

// Глобальные переменные

HINSTANCE hInst;
```

```
TCHAR      szTitle[MAX_LOADSTRING];
TCHAR      szWindowClass[MAX_LOADSTRING];

// Объявление функций этого модуля

ATOM       MyRegisterClass(HINSTANCE hInstance);
BOOL       InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);

extern "C" float* _stdcall maxreal(float *px, int sx);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
  MSG msg;
  HACCEL hAccelTable;

  LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
  LoadString(hInstance, IDC_FINDMAXVALUEINARRAYOFREALS,
             szWindowClass, MAX_LOADSTRING);
  MyRegisterClass(hInstance);

  // Инициализация приложения

  if (!InitInstance (hInstance, nCmdShow))
  {
    return FALSE;
  }

  hAccelTable = LoadAccelerators(hInstance,
                              (LPCTSTR)IDC_FINDMAXVALUEINARRAYOFREALS);

  // Цикл обработки сообщений

  while (GetMessage(&msg, NULL, 0, 0))
```

```
  {
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return (int) msg.wParam;
}

//Функция регистрации класса окна

ATOM MyRegisterClass(HINSTANCE hInstance)
{
  WNDCLASSEX wcex;
  wcex.cbSize          = sizeof(WNDCLASSEX);
  wcex.style           = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc     = (WNDPROC)WndProc;
  wcex.cbClsExtra      = 0;
  wcex.cbWndExtra      = 0;
  wcex.hInstance       = hInstance;
  wcex.hIcon           = LoadIcon(hInstance,
                         (LPCTSTR)IDI_FINDMAXVALUEINARRAYOFREALS);
  wcex.hCursor         = LoadCursor(NULL, IDC_ARROW);

//  wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wcex.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);

  wcex.lpszMenuName  = (LPCTSTR)IDC_FINDMAXVALUEINARRAYOFREALS;
  wcex.lpszClassName = szWindowClass;
  wcex.hIconSm       = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);

  return RegisterClassEx(&wcex);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;
```

```
  hInst = hInstance;
  hWnd  = CreateWindow(szWindowClass, szTitle,
                       WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
                       CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
  if (!hWnd)
  {
    return FALSE;
  }

  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
                         LPARAM lParam)
{
  int wmId, wmEvent;
  PAINTSTRUCT ps;
  HDC hdc;
  char buf[16];
  float xarray[9] = {12.43, 93.54, -23.1, 23.59, 16.09,
                     10.67, -54.7, 11.49, 98.06};
  float *xres;
  int cnt;
  switch (message)
  {
    case WM_COMMAND:
         wmId    = LOWORD(wParam);
         wmEvent = HIWORD(wParam);
         switch (wmId)
         {
           case IDM_ABOUT:
                DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                          hWnd, (DLGPROC)About);
                break;
```

```
          case IDM_EXIT:
               DestroyWindow(hWnd);
               break;
          default:
               return DefWindowProc(hWnd, message, wParam, lParam);

        }
        break;
   case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);

        // Здесь находится код нашего обработчика

        TextOut(hdc, 30, 80, "ARRAY: ", 7);
        for (cnt = 0; cnt < 9; cnt++)
        {
          gcvt(xarray[cnt], 6, buf);
          TextOut(hdc, 100 + cnt*50, 80, buf, 5);
        }
        TextOut(hdc, 30, 100, "MAXIMUM: ", 9);
        xres = maxreal(xarray, 9);
        gcvt(*xres, 5, buf);
        TextOut(hdc, 220, 100, (LPCTSTR)buf, 5);
        EndPaint(hWnd, &ps);
        break;
   case WM_DESTROY:
        PostQuitMessage(0);
        break;
   default:
        return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}

LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
                       LPARAM lParam)
{
  switch (message)
```

```
  {
    case WM_INITDIALOG:
         return TRUE;
    case WM_COMMAND:
         if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
         {
           EndDialog(hDlg, LOWORD(wParam));
           return TRUE;
         }
         break;
  }
  return FALSE;
}
```

Наша программа должна выводить в рабочую область окна приложения две строки. Одна из них должна отображать все элементы массива, а другая, расположенная ниже, отображать на экране значение максимального элемента. Чтобы проделать эти манипуляции, воспользуемся обработчиком сообщения WM_PAINT. Когда приложение получает такое сообщение, это означает, что необходимо выполнить частичную или полную прорисовку окна. Такое сообщение посылает приложению операционная система, например при образовании окна приложения в момент запуска программы. В это время вызывается функция UpdateWindow, которая вынуждает систему отправить сообщение WM_PAINT приложению. В этот момент можно вывести на экран текст при помощи функции TextOut.

В функции обратного вызова WndProc определим следующие переменные:

```
char buf[16];
float xarray[9] = {12.43, 93.54, -23.1, 23.59, 16.09,
                   10.67, -54.7, 11.49, 98.06};
float *xres;
int cnt;
```

Строка buf используется для хранения результата преобразования вещественного числа в текст. Далее определен массив вещественных чисел xarray с 9-ю элементами. Нам понадобится указатель вещественного типа (назовем его xres) и счетчик цикла (cnt) для вывода всех 9-ти элементов на экран.

Следующий программный код в обработчике WM_PAINT выводит две строки на экран (листинг 3.40).

Листинг 3.40. Фрагмент кода из обработчика WM_PAINT оконной процедуры WndProc

```
TextOut(hdc, 30, 80, "ARRAY: ", 7);
for (cnt = 0; cnt < 9; cnt++)
{
  gcvt(xarray[cnt], 6, buf);
  TextOut(hdc, 100 + cnt*50, 80, buf, 5);
}
TextOut(hdc, 30, 100, "MAXIMUM: ", 9);
xres = maxreal(xarray, 9);
gcvt(*xres, 5, buf);
TextOut(hdc, 220, 100, (LPCTSTR)buf, 5);
```

Первая строка обработчика — это функция `TextOut`, принимающая в качестве параметров контекст устройства (`hdc`), горизонтальную и вертикальную координаты в клиентской области окна, указатель на строку текста и количество элементов для вывода на экран.

Поскольку мы не можем непосредственно выводить числа на экран, то необходимо преобразовать наш массив чисел в последовательность строк. Преобразование вещественного числа в последовательность символов можно выполнить при помощи функции `gcvt`, принимающей в качестве параметров вещественное число, количество знаков для вывода и указатель на символьный буфер для хранения результата преобразования. Цикл `for` мы используем для вывода на экран всех 9 элементов массива.

Аналогично выполняется и вывод максимального элемента массива на экран. Но перед этим мы должны вызвать процедуру `maxreal`:

```
xres = maxreal(xarray, 9);
```

Поскольку `xres` — указатель, то для правильной работы функции `gcvt` необходимо передать параметры следующим образом:

```
gcvt(*xres, 5, buf);
```

Максимум после выполнения этого оператора теперь находится в переменной `buf` и отображается на экране функцией `TextOut`.

Необходимо также включить следующее объявление ассемблерной процедуры в текст приложения:

```
extern "C" float* _stdcall maxreal(float *px, int sx);
```

Окно работающего приложения изображено на рис. 3.8.

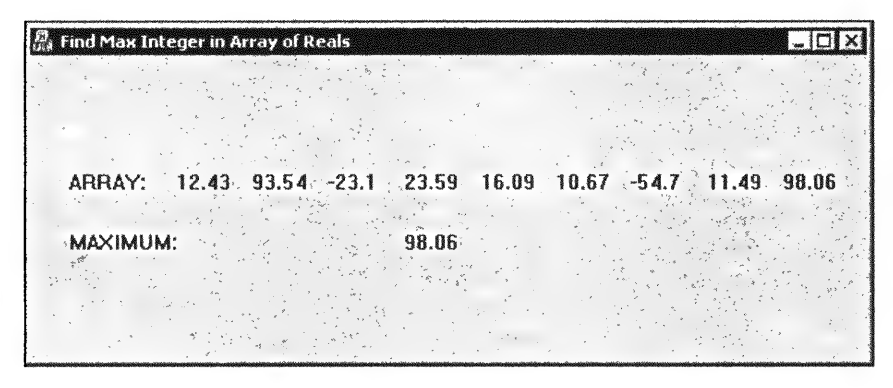


Рис. 3.8. Окно приложения, выполняющего поиск максимума в массиве вещественных чисел

Обычно окно стандартного приложения имеет белый фон. В классе окна он определяется через параметр `wcex.hbrBackground`. Чтобы поменять цвет фона, например, на серый, можно воспользоваться функцией `GetStockObject` с соответствующим параметром. Фрагмент кода демонстрирует, как можно задать другие параметры цвета фона, заменив стандартный белый цвет на серый:

```
//  wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wcex.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);
```

До сих пор мы работали с переменными, представляющими собой числовые значения целого или вещественного типа. Далее мы рассмотрим примеры приложений, выполняющих операции со строками и символьными массивами. Начнем с относительно простых приложений.

В следующем примере необходимо передать в вызывающую программу адрес строки символов и отобразить саму строку в окне приложения.

Процедура на ассемблере представлена в листинге 3.41.

Листинг 3.41. Ассемблерная процедура, передающая строку символов

```
.386
.model flat
  public strshow
.data
  TESTSTR.DB "Hello from subroutine", 0
.code
```

```
strshow proc
  mov     EAX, offset TESTSTR
  ret
strshow endp
end
```

Процедура очень проста. Она возвращает адрес строки `TESTSTR` в регистре `EAX`.

В Delphi-приложении определим внешнюю процедуру `strshow`:

```
implementation
  {$R *.dfm}
  {$L STRSHOW.OBJ}
function strshow: PChar; stdcall; external;
```

Строка из процедуры `strshow` будет отображаться при нажатии любой кнопки мыши в окне приложения. Исходный текст обработчика этого события включает в себя следующий фрагмент кода (листинг 3.42).

Листинг 3.42. Обработчик нажатия кнопки мыши, выводящий строку символов в окно Delphi-приложения

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
                                   Shift: TShiftState; X, Y: Integer);
begin
   Canvas.Font.Height := -20;
   Canvas.Font.Color := clWhite;
   Canvas.TextOut(Rect.Left + 50, Rect.Top + 50, strshow);
end;
```

Для вывода строки текста в окно воспользуемся функцией `TextOut` контекста устройства. В этом обработчике можно задать также параметры шрифта: высоту и цвет.

В секции `var` исходного модуля необходимо определить структуру `Rect`, содержащую координаты клиентской области окна:

```
var
  Form1: TForm1;
  Rect: TRect;
```

Обратите внимание, как определена вызываемая внешняя процедура `strshow`. Она возвращает адрес строки в указателе типа `PChar`. Для обработки строк внешними процедурами необходимо, чтобы они оканчивались нулем, поэтому их нужно объявлять через указатели этого типа.

После запуска вид окна работающего приложения будет выглядеть так, как показано на рис. 3.9.

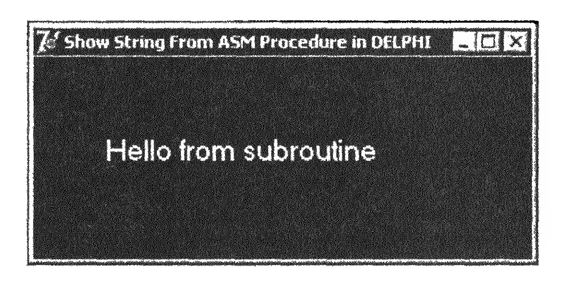


Рис. 3.9. Окно приложения, выполняющего вывод строки из внешней процедуры

Рассмотрим еще один пример, в котором необходимо целиком передать строку из процедуры на ассемблере в основную программу. Для этого скопируем содержимое массива символов вызываемой процедуры в буфер памяти вызывающей программы. Разработаем приложение на Delphi. Процедура на ассемблере (назовем ее `rets`) представлена в листинге 3.43.

Листинг 3.43. Ассемблерная процедура, копирующая строку в основную программу

```
.386
.model flat
  public rets
.data
  TESTSTR   DB  "TEST STRING FROM ASM PROC !", 0
  LENSTR    EQU $-TESTSTR
.code
rets proc
  push    ESI
  push    EDI
  push    EBP
  mov     EBP, ESP
  cld
  mov     ECX, LENSTR
  mov     ESI, offset TESTSTR
```

```
  mov     EDI, DWORD PTR [EBP+16]
  rep     movsb
  pop     EBP
  pop     EDI
  pop     ESI
  ret     4
rets endp
end
```

В качестве параметра процедура получает адрес буфера вызывающей программы, куда нужно скопировать строку. Предполагается, что буфер вызывающей программы имеет достаточный размер для помещения всей строки.

Для копирования строк будем использовать команду ассемблера `movsb`. В регистр `ECX` помещается размер строки в байтах. Регистр `ESI` содержит адрес строки-источника `TESTSTR`, а регистр `EDI` — адрес строки-получателя в основной программе. Копирование осуществляется командой `rep movsb`.

Компиляция ассемблерного модуля выполняется командой:

```
tasm32 /ml rets.asm rets.obj
```

Основная программа на Delphi должна выводить строку из буфера в рабочую область приложения. Сначала опишем переменные, которые использует программа:

```
var
  Form1: TForm1;
  Rect: TRect;
```

Затем объявим внешнюю функцию `rets` в секции `implementation`:

```
implementation
  {$R *.dfm}
  {$L rets.obj}
procedure rets(s1:PChar); stdcall; external;
```

Обратите внимание на то, что процедура в качестве параметра принимает указатель на строку с завершающим нулем. В наших процедурах на ассемблере мы будем в основном использовать строки такого типа.

Разместим на нашей форме кнопку и напишем для нее обработчик нажатия (листинг 3.44).

Листинг 3.44. Обработчик нажатия кнопки в программе на Delphi

```
procedure TForm1.Button1Click(Sender: TObject);
var
   P1: array[0..64] of Char;
begin
   rets(P1);
   Canvas.Font.Color := clYellow;
   Canvas.TextOut(Rect.Left + 20, Rect.Top + 40, P1);
end;
```

Для вывода текста строки в окно приложения мы используем контекст устройства и связанную с ним функцию `TextOut`. Процедуре `rets` в качестве параметра передается адрес строки `P1`. Строка типа `PChar` может быть представлена в виде массива символов, поэтому запись `rets(P1)` является корректной.

Откомпилируем и запустим на выполнение наше приложение. Окно программы представлено на рис. 3.10.

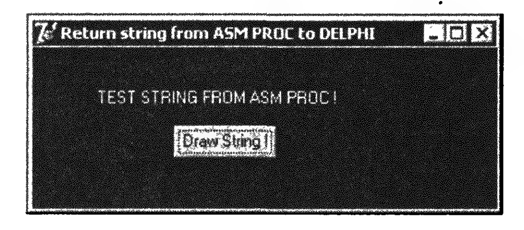


Рис. 3.10. Окно приложения, выполняющего отображение строки из ассемблерной процедуры

Теперь разработаем вариант основной программы для компиляции в Visual C++ .NET. Вначале внесем некоторые изменения в исходный текст ассемблерного модуля из предыдущего примера. Изменения касаются в основном имени процедуры. С учетом сделанных изменений программный код приведен в листинге 3.45.

Листинг 3.45. Ассемблерная процедура, выполняющая копирование строки в буфер памяти основной программы на C++

```
.386
.model flat
  public _rets@4
```

```
.data
  TESTSTR DB  "TEST STRING FROM ASM PROC !",0
  LENSTR  EQU $-TESTSTR
.code
_rets@4   proc
  push    ESI
  push    EDI
  push    EBP
  mov     EBP, ESP
  cld
  mov     ECX, LENSTR
  mov     ESI, offset TESTSTR
  mov     EDI, DWORD PTR [EBP+16]
  rep     movsb
  pop      EBP
  pop     EDI
  pop     ESI
  ret     4
_rets@4   endp
end
```

Наша процедура скопирует строку TESTSTR в область памяти, адрес которой передается из вызывающей программы в качестве параметра. Здесь используется команда movsb, которая скопирует LENSTR число байт из строки TESTSTR, заданной своим смещением в регистре ESI, в строку, заданную смещением в регистре EDI.

Для разработки основной программы воспользуемся Мастером приложений Visual Studio .NET. Выберем диалоговый тип приложения. Разместим на главной форме приложения кнопку Button. Чтобы вывести результирующую строку в окно приложения, воспользуемся обработчиком нажатия кнопки Button. Непосредственный вывод строки выполняет функция TextOut.

Для отображения строки воспользуемся контекстом устройства отображения. Но вначале определим этот объект в обработчике нажатия кнопки:

```
CClientDC dc(this);
```

Структура RECT определяет, как и в приложении на Delphi, координаты клиентской области окна приложения. Чтобы получить текущие координа-

ты, воспользуемся функцией `GetClientRect(&rect)`. Кроме того, необходимо получить саму строку из ассемблерной процедуры и поместить ее в буфер `buf`. Это действие выполнит оператор `rets(buf)`.

Полностью исходный текст обработчика нажатия кнопки приведен в листинге 3.46.

Листинг 3.46. Обработчик нажания кнопки в приложении на C++

```
void CRetunStringFromProcDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  char buf[32];
  rets(buf);
  CClientDC dc(this);
  RECT rect;
  GetClientRect(&rect);
  dc.TextOut((rect.right - rect.left)/5,
             (rect.bottom - rect.top)/ 2,
              buf);
}
```

Буфер принимающей строки `buf` должен иметь достаточный размер, чтобы в нем могла разместиться передаваемая строка.

Окно работающего приложения изображено на рис. 3.11.

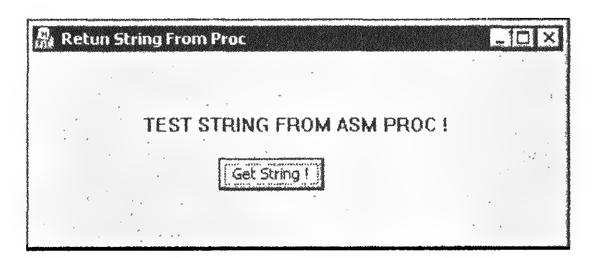


Рис. 3.11. Окно приложения, выполняющего вывод строки из процедуры на ассемблере

При сборке приложения в Visual C++ .NET не забудьте включить объектный модуль на ассемблере в состав проекта. В исходный текст программы после директив `include` необходимо включить объявление внешней процедуры:

```
extern "C" _stdcall rets(char *ps1);
```

Очень часто возникает необходимость передавать в основную программу не целую строку, а лишь ее часть (подстроку), начиная с определенной позиции. Следующий пример приложения показывает, как это можно сделать.

Пусть в ассемблерном модуле находится строка символов, и необходимо передать в вызывающую программу подстроку, начиная с определенной позиции. В этом случае процедура должна получать в качестве параметра величину начального смещения от начала строки. Она должна возвращать в основную программу адрес первого элемента выделенной подстроки.

Исходный текст процедуры на ассемблере (назовем ее `strpart`) приведен в листинге 3.47.

Листинг 3.47. Ассемблерная процедура, возвращающая адрес подстроки

```
.386
.model flat
  public strpart

.data
  TESTSTR DB "Part1 Part2 Part3 Part4 Part5", 0
.code
strpart proc
  push    EBP
  mov     EBP, ESP
  mov     ECX, DWORD PTR [EBP+8]
  mov     EAX, offset TESTSTR
  add     EAX, ECX
  pop     EBP
  ret     4
strpart endp
end
```

Эта процедура в качестве единственного параметра принимает смещение от начала строки. Строка `TESTSTR` состоит из 5 подстрок `"Part1"`, ... , `"Part5"` размером 6 байт каждая (с учетом символа пробела между подстроками). Величина смещения находится в регистре `EBP` по смещению `8` и загружается в регистр `ECX` с помощью команды:

```
mov     ECX, DWORD PTR [EBP+8]
```

В регистр EAX заносится адрес строки из области данных:

```
mov     EAX, offset TESTSTR
```

Адрес первого элемента подстроки вычисляется путем суммирования содержимого регистров EAX и ECX.

Разработаем приложение на Delphi, которое выводило бы на экран исходную строку, подстроку и величину смещения относительно начала исходной строки. Для этого разместим на главной форме приложения три поля редактирования Edit, три метки Label, управляющие стрелки UpDown и кнопку Button. Для большей наглядности свойству Increment компонента UpDown присвоим значение 6 (размер подстроки в строке TESTSTR равен 6 байтам).

Привяжем свойство Position управляющих стрелок UpDown к величине, заданной в поле редактирования Edit3 при помощи свойства Associate. Свойство Min стрелок UpDown установим равным 0, а свойство Max — 30. Полный текст программы приведен в листинге 3.48.

Листинг 3.48. Полный текст программы на Delphi, выводящей на экран подстроку

```
var
  Form1: TForm1;
  StrOFFSET: Integer;
  StrASM: PChar;

implementation
  {$R *.dfm}
  {$L STRPART.OBJ}

function strpart(I1: Integer): PChar; stdcall; external;

procedure TForm1.Button1Click(Sender: TObject);
begin
  StrOFFSET := 0;
  StrASM := strpart(StrOFFSET);
  Edit3.Text := StrASM;
  StrOFFSET := UpDown1.Position;
  StrASM := strpart(StrOFFSET);
```

```
  Edit2.Text := StrASM;
end;
end.
```

На рис. 3.12 изображено окно работающего приложения.

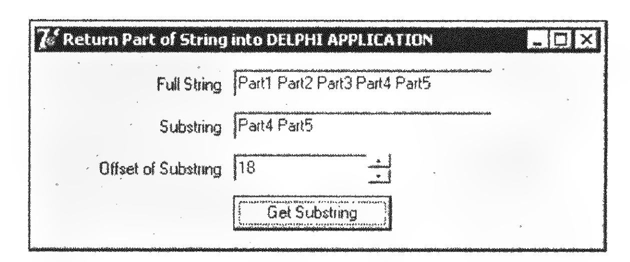


Рис. 3.12. Окно приложения, выполняющего отображение подстроки из ассемблерной процедуры

Во многих случаях требуется обрабатывать строку из Delphi-приложения. Покажем, как это можно сделать при помощи процедуры на ассемблере. Результат возвращается основной программе в виде строки или подстроки. В следующем примере показано, как это сделать. Как и в предыдущей программе, обработку строки будет выполнять процедура на ассемблере, которую мы назовем `strpartd`. Исходный текст процедуры `strpartd` приведен в листинге 3.49.

Листинг 3.49. Ассемблерная процедура, выполняющая обработку строки, находящейся в основной программе

```
.386
.model flat
  public  strpartd
.data
.code
strpartd  proc
  push    EBP
  mov     EBP, ESP
  mov     ECX, DWORD PTR [EBP+12]
  mov     EAX, DWORD PTR [EBP+8]
  add     EAX, ECX
  pop     EBP
  ret     8
strpartd  endp
end
```

Небольшой комментарий к исходному тексту. При вызове процедуры strpartd смещение от начала строки передается первым параметром по значению. Внутри самой процедуры смещение запоминается в регистре ECX. Вторым параметром является адрес строки из основной программы, который мы запоминаем в EAX. Процедура возвращает адрес элемента, следующего за первыми n элементами символьного массива, в регистре EAX. В данном случае начало подстроки будет находиться по адресу, определяемому суммой содержимого регистров EAX и ECX. Поскольку мы используем соглашение stdcall, процедура сама восстанавливает стек командой ret 8.

Исходный текст программы на Delphi приведен в листинге 3.50.

Листинг 3.50. Программа на Delphi, выполняющая отображение подстроки при помощи ассемблерной процедуры

```
var
  Form1: TForm1;
  S1: array [1..16] of Char = 'SOURCE STRING!!'#0;
  S1Offset: Integer;

implementation
  {$R *.dfm}
  {$L STRPARTD.OBJ}

function strpartd(ps1: PChar; I1: Integer): PChar; stdcall; external;

procedure TForm1.FormCreate(Sender: TObject);
begin
  S1Offset := 0;
  Edit1.Text := S1;
  Edit2.Text := strpartd(@S1, S1Offset);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  S1Offset := UpDown1.Position;
  Edit2.Text := strpartd(@S1, S1Offset);
end;
```

Как видно из листинга основной программы, переменная `S1` представляет собой строку с завершающим нулем, причем при определении размера такой строки необходимо учитывать и нулевой символ. Оператор:

```
Edit2.Text := strpartd(@S1, S1Offset);
```

присваивает свойству `Text` поля редактирования `Edit2` адрес подстроки, которая отображается на экране. Окно работающего приложения изображено на рис. 3.13.

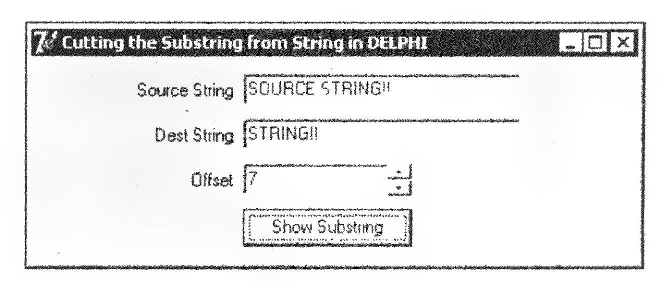


Рис. 3.13. Окно приложения, выполняющего отображение подстроки основной программы

Хочется напомнить некоторые очень важные детали, касающиеся работы со строковыми переменными. Манипуляции со строками и символьными массивами очень часто встречаются в программистской практике. К сожалению, в документации фирмы Borland работа со строками описана плохо. В программах на Delphi во многих случаях требуется обрабатывать строки стандартного типа — строки с завершающим нулем. В нашей программе мы также используем строку такого типа. Обратите внимание на ее объявление:

```
S1: array [1..16] of Char = 'SOURCE STRING!!'#0;
```

Для манипуляций с такой строкой из внешних процедур требуется использование указателей типа `PChar`. Объявление строки `S1` как массива символов с завершающим нулем как раз и обеспечивает такую возможность. При этом переменная-указатель типа `PChar` содержит адрес первого элемента строки, т. е. `S1[1]`.

Необходимо помнить, что компилятор Delphi не производит проверку диапазона изменения индекса, поэтому, например, элемент `S1[2]` — это следующий элемент массива, а элемент `S1[0]` — предыдущий. Полная длина строки должна учитывать наличие нулевого символа в конце строки `#0`. Здесь необходимо быть очень внимательным.

Чтобы отобразить такую строку в поле редактирования Edit1, нужно просто выполнить оператор:

```
Edit1.Text := S1;
```

Посмотрим, как будет выглядеть эта же программа, возвращающая часть строки, на C++ .NET. Необходимо сделать некоторые изменения в исходном тексте ассемблерной процедуры. Они связаны с изменением имени в соответствии с требованиями компилятора Visual C++ для директивы stdcall (листинг 3.51).

Листинг 3.51. Ассемблерная процедура, возвращающая подстроку в программу на C++

```
.386
.model flat
  public  _strpart@8
.data
.code
_strpart@8 proc
  push    EBP
  mov     EBP, ESP
  mov     ECX, DWORD PTR [EBP+12]
  mov     EAX, DWORD PTR [EBP+8]
  add     EAX, ECX
  pop     EBP
  ret     8
_strpart@8 endp
end
```

В качестве шаблона для основной программы на C++ .NET выберем классический вариант процедурно-ориентированного Windows-приложения. После генерации каркаса Мастером приложений сделаем некоторые изменения и дополнения в исходном тексте и добавим в меню пункт Return Part of String. Свяжем с ним идентификатор ID_PartStr. При выборе этого пункта меню в окне работающего приложения будут отображаться исходная строка, подстрока и смещение в исходной строке.

В разделе объявлений основной программы WinMain сделаем ссылку на внешнюю процедуру:

```
extern "C" char* _stdcall strpart(char *ps, int off);
```

В качестве параметров процедура strpart принимает адрес исходной строки и смещение от ее начала.

Определим также переменные, которые используются нашим приложением:

```
char src[] = "STRING1 STRING2 STRING3 STRING4 STRING5";
char *dst;
int  off, ioff;
char buf[4];
```

где:

- строка src — исходная строка для обработки;
- строка dst — строка-получатель;
- целочисленная переменная off определяет смещение от начала исходной строки;
- строка buf и целое ioff используются функцией sprintf для форматирования вывода.

В функции обратного вызова WndProc напишем обработчик выбора пункта меню ID_PartStr (листинг 3.52).

Листинг 3.52. Обработчик выбора пункта меню

```
case ID_PartStr:
  hdc = GetDC(hWnd);
  GetClientRect(hWnd, &rect);
  off = 10;
  dst = strpart(src, off);
  ioff = sprintf(buf,"%d",off);
  TextOut(hdc,(rect.right - rect.left)/4, (rect.bottom - rect.top)/4,
          "Source:", 7);
  TextOut(hdc, (rect.right - rect.left)/3, (rect.bottom - rect.top)/4,
          src, strlen(src));
  TextOut(hdc, (rect.right - rect.left)/4, (rect.bottom - rect.top)/3,
          "Dest:", 5);
  TextOut(hdc, (rect.right - rect.left)/3, (rect.bottom - rect.top)/3,
          dst, strlen(dst));
  TextOut(hdc, (rect.right - rect.left)/4,
                (rect.bottom - rect.top)/3 + 30, "Offset:", 7);
  TextOut(hdc, (rect.right - rect.left)/3,
```

```
                (rect.bottom - rect.top)/3 + 30, buf, ioff);
  ReleaseDC(hWnd, hdc);
  break;
```

Для вывода текста в клиентскую область окна мы используем, как и в других примерах, функцию `TextOut`. В качестве первого параметра она получает дескриптор контекста устройства отображения для рисования на экране дисплея. Дескриптор контекста возвращается функцией `GetDC`.

Поскольку нам нужно, чтобы выводимый текст попадал в рабочую область окна, то желательно получить координаты этой области с помощью функции `GetClientRect`. Можно обойтись и без нее, но тогда придется повозиться с расположением текста в окне приложения.

Для форматирования вывода целочисленной переменной `off` на экран мы используем функцию `sprintf`. Так как прототип этой функции описан в файле stdio.h, то необходимо в раздел деклараций функции `WinMain` включить запись:

```
#include <stdio.h>
```

И, наконец, чтобы наше окно лучше смотрелось, заменим стандартный белый цвет фона на серый:

```
//  wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wcex.hbrBackground = (HBRUSH)GetStockObject(GRAY_BRUSH);
```

Полный исходный текст нашего приложения приведен в листинге 3.53.

Листинг 3.53. Текст программы на С++, выводящей подстроку в окно приложения

```
// Точка входа в программу
#include "stdafx.h"
#include "Return Part of String in C.NET.h"
#define  MAX_LOADSTRING 100
#include <stdio.h>

HINSTANCE hInst;
TCHAR     szTitle[MAX_LOADSTRING];
TCHAR     szWindowClass[MAX_LOADSTRING];
```

```
// Ссылки на функции, определенные в этом модуле

ATOM    MyRegisterClass(HINSTANCE hInstance);
BOOL    InitInstance(HINSTANCE, int);
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK  About(HWND, UINT, WPARAM, LPARAM);

extern "C" char* _stdcall strpart(char *ps, int off);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
  MSG msg;
  HACCEL hAccelTable;

  LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
  LoadString(hInstance, IDC_RETURNPARTOFSTRINGINCNET,
             szWindowClass, MAX_LOADSTRING);
  MyRegisterClass(hInstance);
  if (!InitInstance(hInstance, nCmdShow))
  {
    return FALSE;
  }

  hAccelTable = LoadAccelerators(hInstance,
                (LPCTSTR)IDC_RETURNPARTOFSTRINGINCNET);

  while (GetMessage(&msg, NULL, 0, 0))
  {
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return (int)msg.wParam;
}
```

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{
  WNDCLASSEX wcex;
  wcex.cbSize          = sizeof(WNDCLASSEX);
  wcex.style           = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc     = (WNDPROC)WndProc;
  wcex.cbClsExtra      = 0;
  wcex.cbWndExtra      = 0;
  wcex.hInstance       = hInstance;
  wcex.hIcon           = LoadIcon(hInstance,
                                (LPCTSTR)IDI_RETURNPARTOFSTRINGINCNET);
  wcex.hCursor         = LoadCursor(NULL, IDC_ARROW);
  wcex.hbrBackground   = (HBRUSH)GetStockObject(GRAY_BRUSH);
  //wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wcex.lpszMenuName    = (LPCTSTR)IDC_RETURNPARTOFSTRINGINCNET;
  wcex.lpszClassName   = szWindowClass;
  wcex.hIconSm         = LoadIcon(wcex.hInstance,
                                (LPCTSTR)IDI_SMALL);
  return RegisterClassEx(&wcex);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;
  hInst = hInstance;
  hWnd  = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL,
                       hInstance, NULL);
  if (!hWnd)
  {
    return FALSE;
  }
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
                         LPARAM lParam)
```

```
{
  int  wmId, wmEvent;
  PAINTSTRUCT ps;
  HDC  hdc;
  RECT rect;
  char src[] = "STRING1 STRING2 STRING3 STRING4 STRING5";

  char *dst;
  int  off, ioff;
  char buf[4];

  switch (message)
  {
    case WM_COMMAND:
         wmId    = LOWORD(wParam);
         wmEvent = HIWORD(wParam);
         switch (wmId)
         {
           case IDM_ABOUT:
                DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
                          hWnd, (DLGPROC)About);
                break;
           case IDM_EXIT:
                DestroyWindow(hWnd);
                break;
           case ID_PartStr:
                hdc  = GetDC(hWnd);
                GetClientRect(hWnd, &rect);
                off  = 10;
                dst  = strpart(src, off);
                ioff = sprintf(buf,"%d",off);
                TextOut(hdc, (rect.right - rect.left)/4,
                       (rect.bottom - rect.top)/4, "Source:", 7);
                TextOut(hdc, (rect.right - rect.left)/3,
                       (rect.bottom - rect.top)/4, src, strlen(src));
                TextOut(hdc, (rect.right - rect.left)/4,
                       (rect.bottom - rect.top)/3, "Dest:", 5);
                TextOut(hdc, (rect.right - rect.left)/3,
```

```
                    (rect.bottom - rect.top)/3, dst, strlen(dst));
                TextOut(hdc, (rect.right - rect.left)/4,
                    (rect.bottom - rect.top)/3 + 30, "Offset:", 7);
                TextOut(hdc, (rect.right - rect.left)/3,
                    (rect.bottom - rect.top)/3 + 30, buf, ioff);

                ReleaseDC(hWnd, hdc);
                break;
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}

LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
                       LPARAM lParam)
{
  switch (message)
  {
    case WM_INITDIALOG:
        return TRUE;
    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
          EndDialog(hDlg, LOWORD(wParam));
          return TRUE;
        }
```

```
        break;
    }
    return FALSE;
}
```

На рис. 3.14 изображено окно работающего приложения.

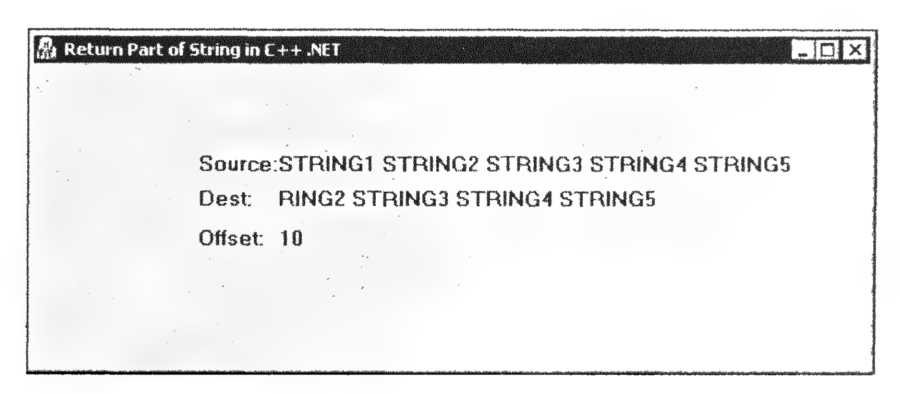


Рис. 3.14. Окно приложения, выполняющего отображение части строки из программы на C++ .NET

Очень часто на практике возникает необходимость в поиске какого-либо элемента строки. Приведенный далее пример демонстрирует, как это можно реализовать.

Пусть имеется строка символов, в которой ищется какой-либо символ. Необходимо в вызывающую программу вернуть порядковый номер первого встретившегося элемента строки, если таковой найден, или 0 в случае неудачи. В ассемблерную процедуру в качестве параметров передаются адрес строки и символ, который необходимо найти.

Исходный текст процедуры (назовем ее `charpos`) на ассемблере для вызова из Delphi-приложения представлен в листинге 3.54.

Листинг 3.54. Ассемблерная процедура, выполняющая поиск символа в строке, находящейся в программе на Delphi

```
.386
.model flat
  public charpos
.data
.code
charpos proc
  push    EBX
```

```
  push    EBP
  mov     EBP, ESP
  mov     EBX, DWORD PTR [EBP+12]
  xor     EAX, EAX
  mov     AL,  BYTE PTR [EBP+16]
  mov     ECX, 1
next_check:
  cmp     AL, [EBX]
  je      quit
  cmp     BYTE PTR [EBX], 0
  jne     inc_cnt
  jmp     not_found
quit:
  mov     EAX, ECX
  pop     EBP
  pop     EBX
  ret     8
inc_cnt:
  inc     ECX
  inc     EBX
  jmp     next_check
not_found:
  xor     ECX, ECX
  jmp     quit
charpos endp
end
```

Наша процедура в качестве параметров принимает адрес строки `ps1` и символ `c1`, который мы ищем. Процедура возвращает номер позиции в строке, где впервые встречается этот элемент. Первый элемент строки имеет номер позиции 1, поэтому счетчику позиции мы присваиваем начальное значение с помощью команды:

```
mov    ECX, 1
```

Адрес строки мы помещаем в регистр `EBX`, а символ, который ищем — в регистр `AL`. Сравнение символа по адресу в регистре `EBX` с искомым симво-

лом в AL и возможные варианты продолжения выполняются следующими командами:

```
cmp     AL, [EBX]
je      quit
cmp     BYTE PTR [EBX], 0
jne     inc_cnt
jmp     not_found
```

Если искомый символ в строке найден, в счетчик ECX записывается его порядковый номер, если же последним элементом строки является ноль, т. е. строка закончилась, в ECX записывается 0. Если символ не обнаружен и строка еще не закончилась, инкрементируем адрес в регистре EBX, счетчик — в регистре ECX и возвращаемся в начало цикла командой:

```
jmp     next_check
```

Процедура возвращает значение, как обычно, в регистре EAX и освобождает стек командой ret 8.

Наше приложение на Delphi 7 является более сложным, чем предыдущие примеры. Вначале на главной форме приложения разместим необходимые визуальные компоненты: меню MainMenu, список ListBox, три поля редактирования Edit, три метки Label и кнопку Button. В меню добавим три пункта: Choose Item, Search! и Exit. При выборе пункта меню Choose Item из списка ListBox выбирается строка и помещается в поле Edit1 (метка Selected). Далее при выборе пункта Search! символ, помещенный в поле редактирования Edit2 (метка Char to search), ищется в выбранной строке. Если символ найден, то в поле редактирования Edit3 (метка Number) отображается его порядковый номер в строке, иначе выдается сообщение "Character not found!".

Перейдем к анализу исходного текста Delphi-приложения. В секции implementation объявим ассемблерную процедуру charpos:

```
implementation
  {$R *.dfm}
  {$L f:\asm\tasm\charpos.obj}
function charpos(ps: PChar; c1: Char): Integer; stdcall; external;
```

В секции объявления переменных укажем массив строк, из которого будут выбираться элементы:

```
var
  Form1: TForm1;
  sarray: array[1..5] of string[7] = ('First', 'Second', 'Third',
                                      'Fourth', 'Fifth');
```

Далее напишем обработчики пунктов меню и процедуру инициализации (листинг 3.55).

Листинг 3.55. Обработчики пунктов меню и процедура инициализации на Delphi

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Index: Integer;
begin
  for Index := 1 to 5 do
  begin
    ListBox1.ItemIndex := Index;
    ListBox1.Items.Add(sarray[Index]);
  end;
end;

procedure TForm1.ChooseItemClick(Sender: TObject);
var
  nItem: Integer;
begin
  nItem := ListBox1.ItemIndex;
  Edit1.Text := ListBox1.Items[nItem];
end;

procedure TForm1.SearchClick(Sender: TObject);
var
  ps, ps1: PChar;
  c1: Char;
  num: Integer;
```

```
begin
  ps := PChar(Edit1.Text);
  ps1 := PChar(Trim(Edit2.Text));
  c1 := ps1[0];
  num := charpos(ps, c1);
  if num = 0 then
    Edit3.Text := 'Character not found!'
  else
    Edit3.Text := IntToStr(num);
end;

procedure TForm1.ExitClick(Sender: TObject);
begin
  Close();
end;
```

Окно работающего приложения изображено на рис. 3.15.

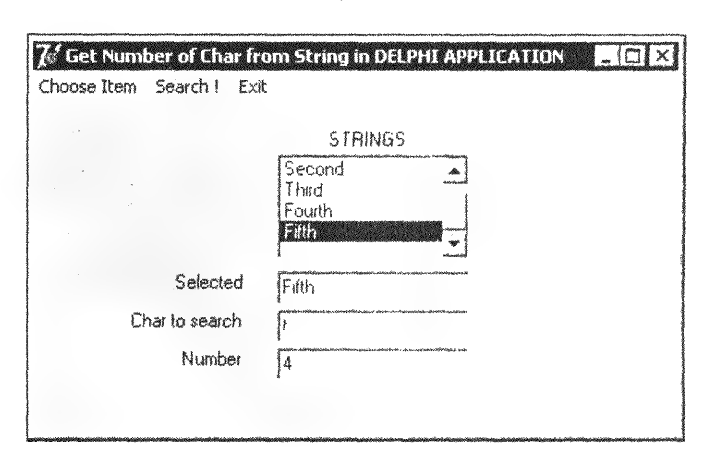


Рис. 3.15. Окно приложения, выполняющего поиск позиции символа в строке

Разработаем вариант реализации этого примера на языке Visual C++ .NET. Первое, что необходимо сделать — это изменить имя нашей процедуры в ассемблерном модуле в соответствии с директивой `stdcall`.

Исходный текст процедуры будет выглядеть так, как приведено в листинге 3.56.

Листинг 3.56. Ассемблерная процедура поиска символа в строке для работы с программой на C++

```
.386
.model flat
  public _charpos@8
.data
.code
_charpos@8 proc
  push    EBX
  push    EBP
  mov     EBP, ESP
  mov     EBX, DWORD PTR [EBP+12]
  xor     EAX, EAX
  mov     AL,  BYTE PTR [EBP+16]
  mov     ECX, 1
next_check:
  cmp     AL, [EBX]
  je      quit
  cmp     BYTE PTR [EBX], 0
  jne     inc_cnt
  jmp     not_found
quit:
  mov     EAX, ECX
  pop     EBP
  pop     EBX
  ret     8
inc_cnt:
  inc     ECX
  inc     EBX
  jmp     next_check
not_found:
  xor     ECX, ECX
  jmp     quit
_charpos@8 endp
end
```

В качестве шаблона для С++ приложения выберем диалоговое окно. Разместим на главной форме приложения три поля редактирования `Edit`, три элемента `Static Text` и кнопку `Button`. Свяжем с полями редактирования `Source` и `Character` переменные `src` и `cSrc` типа `CString`, а с полем редактирования `Number` — переменную `iPos` целочисленного типа. Напишем обработчик события для нажатия кнопки `Button` (листинг 3.57).

Листинг 3.57. Обработчик нажатия кнопки в приложении на С++

```
void GetNumberOfCharinStringforCNETDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  CString s1;
  CString c1;
  char *pc1;
  UpdateData(TRUE);
  s1 = src;
  c1 = cSrc;
  pc1 = c1.GetBuffer(8);
  iPos = charpos(s1.GetBuffer(16), *pc1);
  UpdateData(FALSE);
}
```

В случае если символ найден, то в поле редактирования `Number` будет выведен номер элемента, иначе выводится 0.

На рис. 3.16 представлено окно работающего приложения.

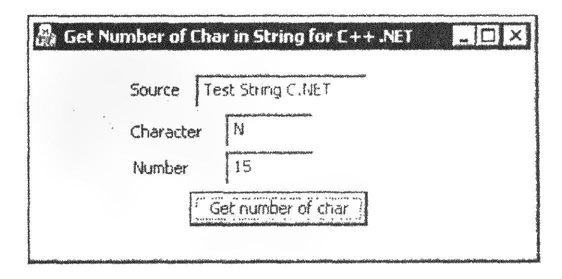


Рис. 3.16. Окно приложения, выполняющего поиск позиции символа в строке

3.4. Сравнительный анализ программного кода на ассемблере и C++

Эффективность применения процедур на ассемблере можно проиллюстрировать на последнем примере. В качестве тестового приложения выберем простую программу копирования одной строки в другую. Для начала напишем программу на "чистом" Visual C++ .NET. Создадим приложение на основе диалогового окна. Разместим два поля редатирования `Edit` (`Source` и `Dest`) и кнопку `Button`. Назовем кнопку `Copy Strings with C++ .NET`. Поставим в соответствие элементам `Edit` переменные `cSrc` и `cDst` типа `CString`. Обработчик нажатия кнопки выполнит копирование одной строки в другую при помощи операторов языка C++ (листинг 3.58).

Листинг 3.58. Копирование одной строки в другую с помощью операторов C++ в обработчике нажатия кнопки

```
void CCOPYSTRINGCNETDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  char src[25] = "NET VARIANT FOR COPYING!";
  char dst[25];
  int srcLen = sizeof(src);
  for (int cnt = 0; cnt < srcLen; cnt++)
    dst[cnt] = src[cnt];
  cSrc = (CString)src;
  cDst = (CString)dst;
  UpdateData(FALSE);
}
```

Добавим еще одну кнопку на главную форму приложения и назовем ее `Copy Strings with ASM PROC`. Для обработчика нажатия этой кнопки разработаем и откомпилируем ассемблерную процедуру (назовем ее `copystr`) (листинг 3.59).

Листинг 3.59. Копирование одной строки в другую с использованием ассемблерной процедуры

```
.386
.model flat
  public _copystr@12
.data
```

```
.code
_copystr@12 proc
  push    ESI
  push    EDI
  push    EBP
  mov     EBP, ESP
  mov     ESI, DWORD PTR [EBP+20]
  mov     EDI, DWORD PTR [EBP+16]
  mov     ECX, DWORD PTR [EBP+24]
  cld
  rep     movsb
  pop     EBP
  pop     EDI
  pop     ESI
  ret     12
_copystr@12 endp
end
```

Используем эту процедуру для копирования строк. В качестве параметров процедура принимает адрес исходной строки `EBP+20`, адрес строки назначения `EBP+16` и размер строки `EBP+24`. Копирование выполняется при помощи команды `movsb` с префиксом повторения, равным размеру исходной строки.

Сам обработчик нажатия кнопки `Copy Strings with ASM PROC` представлен в листинге 3.60.

Листинг 3.60. Обработчик нажатия кнопки `Copy Strings with ASM PROC` в программе на C++

```
void CCOPYSTRINGCNETDlg::OnBnClickedButton2()
{
  // TODO: Add your control notification handler code here

  char src[25] = "ASM VARIANT FOR COPYING!";
  char dst[25];
  int srcLen = sizeof(src);
  copystr(dst, src, srcLen);
  cSrc = (CString)src;
  cDst = (CString)dst;
  UpdateData(FALSE);
}
```

Кроме того, объявим нашу внешнюю ассемблерную процедуру в разделе деклараций:

```
extern "C" _stdcall copystr(char *dst, char *src, int len);
```

Добавим объектный модуль `copystr` в проект. После компиляции и сборки программы запустим ее на выполнение. При нажатии кнопки строки скопируются при помощи программного кода, разработанного средствами языка C++, и мы увидим результат в окне приложения (рис. 3.17).

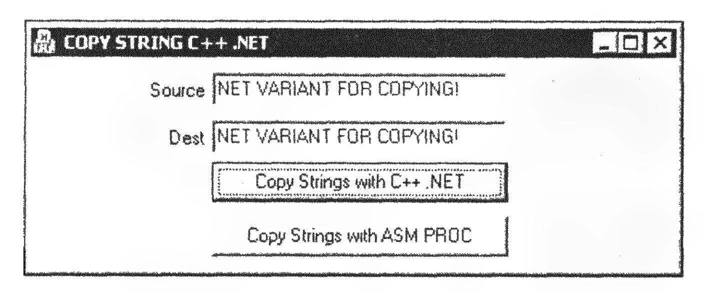


Рис. 3.17. Окно приложения, выполняющего копирование одной строки в другую при помощи операторов C++

При нажатии кнопки `Copy Strings with ASM PROC` строки копируются с помощью внешней процедуры на ассемблере `copystr`.

Окно приложения изменится, как на рис.3.18.

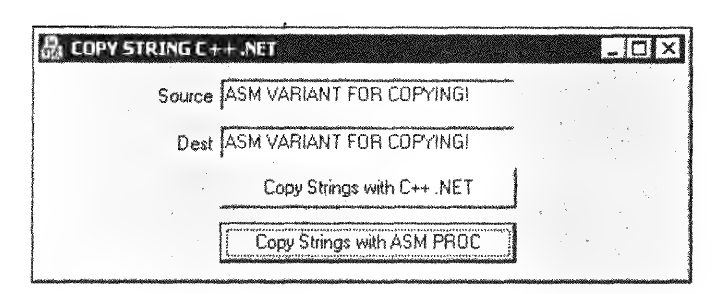


Рис. 3.18. Окно приложения, выполняющего копирование строк при помощи ассемблерной процедуры

Процедура `copystr` довольно проста и использует для копирования практически одну команду `rep movsb`.

Для анализа процедуры копирования с использованием только языка C++ проведем отладку приложения. Напомним себе, что мы будем анализиро-

вать следующий фрагмент кода в обработчике нажатия кнопки Copy Strings with C++ .NET:

```
int srcLen = sizeof(src);
for (int cnt = 0; cnt < srcLen; cnt++)
  dst[cnt] = src[cnt];
```

Код, сгенерированный отладчиком для C++-варианта, приведен в листинге 3.61.

Листинг 3.61. Ассемблерный код обработчика из листинга 3.58

```
       char src[25] = "NET VARIANT FOR COPYING!";
00413678 mov      ecx,6
0041367D mov      esi,offset string "NET VARIANT FOR COPYING!" (4235F8h)
00413682 lea      edi,[src]
00413685 rep movs dword ptr [edi],dword ptr [esi]
00413687 movs     byte ptr [edi],byte ptr [esi]

       char dst[25];
       int srcLen = sizeof(src);
00413688 mov      dword ptr [srcLen],19h
       for (int cnt = 0;cnt < srcLen;cnt++)
0041368F mov      dword ptr [cnt],0
00413696 jmp      CCOPYSTRINGCNETDlg::OnBnClickedButton1+61h (4136A1h)
00413698 mov      eax,dword ptr [cnt]
0041369B add      eax,1
0041369E mov      dword ptr [cnt],eax
004136A1 mov      eax,dword ptr [cnt]
004136A4 cmp      eax,dword ptr [srcLen]
004136A7 jge      CCOPYSTRINGCNETDlg::OnBnClickedButton1+79h (4136B9h)

       dst[cnt] = src[cnt];
004136A9 mov      eax,dword ptr [cnt]
004136AC mov      ecx,dword ptr [cnt]
004136AF mov      dl,byte ptr src[ecx]
004136B3 mov      byte ptr dst[eax],dl
004136B7 jmp      CCOPYSTRINGCNETDlg::OnBnClickedButton1+58h (413698h)

       cSrc = (CString)src;
```

```
004136B9 lea      eax,[src]
004136BC push     eax
004136BD lea      ecx,[ebp-140h]
004136C3 call  ATL::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char>
> >::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char> > > (4118C0h)
004136C8 mov      dword ptr [ebp-154h],eax
004136CE mov      ecx,dword ptr [ebp-154h]
004136D4 mov      dword ptr [ebp-158h],ecx
004136DA mov      dword ptr [ebp-4],0
004136E1 mov      edx,dword ptr [ebp-158h]
004136E7 push     edx
004136E8 mov      ecx,dword ptr [this]
004136EB add      ecx,78h
004136EE call ATL::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char>
> >::operator= (411983h)
004136F3 mov      dword ptr [ebp-4],0FFFFFFFFh
004136FA lea      ecx,[ebp-140h]
00413700 call ATL::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char>
> >::~CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char> > > (411190h)

       cDst = (CString)dst;
00413705 lea      eax,[dst]
00413708 push     eax
00413709 lea      ecx,[ebp-14Ch]
0041370F call ATL::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char>
> >::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char> > > (4118C0h)
00413714 mov      dword ptr [ebp-154h],eax
0041371A mov      ecx,dword ptr [ebp-154h]
00413720 mov      dword ptr [ebp-158h],ecx
00413726 mov      dword ptr [ebp-4],1
0041372D mov      edx,dword ptr [ebp-158h]
00413733 push     edx
00413734 mov      ecx,dword ptr [this]
00413737 add      ecx,7Ch
0041373A call ATL::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char>
> >::operator= (411983h)
0041373F mov      dword ptr [ebp-4],0FFFFFFFFh
00413746 lea      ecx,[ebp-14Ch]
0041374C call ATL::CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char>
> >::~CStringT<char,StrTraitMFC<char,ATL::ChTraitsCRT<char> > > (411190h)
```

```
      UpdateData(FALSE);
        push      0
00413753 mov      ecx,dword ptr [this]
00413756 call     CWnd::UpdateData (4111AEh)
```

В дизассемблированном листинге наибольший интерес представляет фрагмент кода, выполняющий процедуру копирования строк в цикле `for` (листинг 3.62).

Листинг 3.62. Фрагмент кода на ассемблере, соответствующий циклу for

```
      int srcLen = sizeof(src);
00413688 mov      dword ptr [srcLen],19h

      for (int cnt = 0;cnt < srcLen;cnt++)
0041368F mov      dword ptr [cnt],0
00413696 jmp      CCOPYSTRINGCNETDlg::OnBnClickedButton1+61h (4136A1h)
00413698 mov      eax,dword ptr [cnt]
0041369B add      eax,1
0041369E mov      dword ptr [cnt],eax
004136A1 mov      eax,dword ptr [cnt]
004136A4 cmp      eax,dword ptr [srcLen]
004136A7 jge      CCOPYSTRINGCNETDlg::OnBnClickedButton1+79h (4136B9h)

      dst[cnt] = src[cnt];
004136A9 mov      eax,dword ptr [cnt]
004136AC mov      ecx,dword ptr [cnt]
004136AF mov      dl,byte ptr src[ecx]
004136B3 mov      byte ptr dst[eax],dl
004136B7 jmp      CCOPYSTRINGCNETDlg::OnBnClickedButton1+58h (413698h)
```

Даже беглый взгляд на фрагмент дизассемблированного кода для цикла `for` позволяет сделать некоторые выводы. Как видите, компилятор генерирует избыточный код (даже если опция оптимизации включена!). Это мало сказывается на быстродействии программы, если мы обрабатываем несколько символов в строке. Однако чаще всего приходится обрабатывать большие массивы данных или строк, и замедление работы программы станет более существенным.

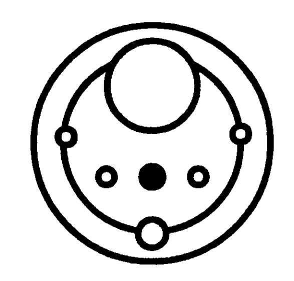
Еще одно замечание касается использования команд переходов. Архитектура современных процессоров основана на алгоритмах *упреждающей выборки*

(prefetch) и *прогнозирования* (prediction) очереди команд. Команды переходов, генерируемые компилятором, часто не учитывают эту особенность и замедляют работу программы. Подобные нюансы оптимизации работы на уровне процессора частично реализованы в компиляторе C++ фирмы Intel.

Как видим, использование ассемблера даже в такой небольшой программе может дать выигрыш в быстродействии.

На этом мы закончим рассмотрение интерфейсов программ на языке ассемблера с программами на языках высокого уровня. Все примеры из этой главы могут быть легко модифицированы читателями для использования в собственных разработках. Можно надеяться, что программисты, пишущие на ассемблере, оценят мощь языков высокого уровня и захотят применить их в своих разработках. Можно также надеяться, что программисты, пишущие на языках высокого уровня, откроют для себя легкость и изящество ассемблерных модулей и будут использовать их для оптимизации своих приложений.

Глава 4



Программирование приложений в Windows на языке ассемблера: первые шаги

Эта глава целиком посвящена вопросам разработки Windows-приложений на языке ассемблера. Мы рассмотрим программы, полностью написанные на ассемблере. Потребность в написании таких программ возникает, когда нужно спроектировать максимально быстрое приложение, например для мобильных систем, или же необходимо максимально увеличить производительность какого-либо устройства.

Вначале вспомним немного теории. 32-разрядные Windows-приложения, как известно, выполняются в защищенном режиме и имеют в своем распоряжении 4-гигабайтное пространство адресов. Однако это совсем не означает, что программа будет использовать его полностью.

В отличие от 16-разрядных приложений, где все программы могли "видеть" друг друга, для 32-разрядных приложений все по-другому. Каждое такое приложение выполняется в своем собственном пространстве адресов, принадлежащих только ему. Это исключает риск потери данных или сбоя программы, что возможно в случае 16-разрядных приложений, где другая программа могла нарушить ход выполнения приложения.

И еще одно отличие. В 16-разрядных приложениях применялась адресация с использованием сегментных регистров CS, DS и ES. Это влекло за собой определенные неудобства. Во-первых, сегменты кода программы и данных не могли превышать 64 Кбайт, и для компоновки больших приложений приходилось применять различные ухищрения. В 32-разрядных приложениях не существует разделения на сегменты, и программисту не нужно заботиться об адресации сегментных регистров. Во-вторых, вычисление исполнительных адресов в 16-разрядных приложениях (сегмент + смещение) замедляло работу программы.

Графические приложения Windows основаны на архитектуре, управляемой событиями. Я не буду подробно останавливаться на особенностях работы приложений в операционных системах Windows, т. к. имеется много литера-

туры, достаточно хорошо описывающей ключевые аспекты данного вопроса. Следует упомянуть лишь основные функциональные составляющие графических Windows-приложений.

Программный код такого приложения должен включать в себя *оконную процедуру*. Оконная процедура регистрируется в системе и вызывается всякий раз, когда выполняется какая-либо операция над окном приложения. Приложение обязательно должно включать в себя и *цикл обработки сообщений*. В этом цикле программа выбирает из очереди сообщения, предназначенные для окна приложения, и направляет их в Windows. Операционная система передает сообщения оконной процедуре, которая и выполняет соответствующие действия. Кроме того, в программе обычно присутствуют и операторы, выполняющие инициализацию приложения.

Программный код стандартного графического Windows-приложения обычно размещается в функции `WinMain`, выполняющей следующие операции:

- инициализация приложения;
- регистрация класса окна приложения;
- инициализация цикла обработки сообщений;
- завершение выполнения приложения.

Теперь можно перейти к программной реализации оконного Windows-приложения. Прежде всего проанализируем программный код такого приложения на одном из языков высокого уровня. Такой подход позволяет лучше понять структуру приложения Windows. Можно воспользоваться, например, Мастером приложений Visual C++ .NET и сгенерировать Win32 Project. Даже если читатель не знаком с языком C++, он все равно легко поймет исходный текст такого приложения.

Наша базовая программа на C++ выводит пустое окно на экран. Изменим программу так, чтобы она выводила в рабочую область окна приложения текст, например `"HELLO FROM VISUAL C++ .NET!"`. Для начала упростим исходный текст приложения, сгенерированного для нас Мастером приложений C++ .NET, исключив обработчик меню из программы.

Работу с графикой и текстом мы будем рассматривать подробно в *главе 5*, а сейчас нам понадобятся некоторые основные понятия. В системе Windows можно выводить информацию только в рабочую область окна приложения. Чтобы вывести текст, воспользуемся следующей особенностью системы: при любых изменениях размеров окна, при восстановлении рабочей области, а также при инициализации окна операционная система Windows посылает приложению сообщение `WM_PAINT`. Оконная процедура программы может (хотя и не обязательно) использовать это сообщение, чтобы перерисовать рабочую область окна. Такая перерисовка также восстанавливает или рисует текст в рабочей области.

Но как получить от Windows сообщение WM_PAINT? Один из способов — вызвать функцию UpdateWindow перед входом в цикл обработки сообщений. Другой возможный вариант — вызвать функцию InvalidateRect. В нашем приложении используется функция UpdateWindow, которая вынуждает операционную систему генерировать сообщение WM_PAINT.

Нам необходимо сделать лишь некоторые изменения в исходном тексте программы, а именно — в обработчике сообщения WM_PAINT оконной процедуры WndProc. Для того чтобы вывести текст в окно приложения, воспользуемся функцией TextOut в обработчике сообщения WM_PAINT. В качестве параметров эта функция принимает дескриптор контекста устройства отображения, начальные координаты выводимого текста, адрес и размер строки текста. Контекст устройства в операционных системах Windows представляет собой структуру данных, в которой содержится описание графических атрибутов таких устройств, как дисплей или принтер. Контекст устройства позволяет разрабатывать аппаратно-независимый графический интерфейс пользователя.

Фрагмент кода обработчика сообщения WM_PAINT в результате таких модификаций будет выглядеть так, как показано в листинге 4.1.

Листинг 4.1. Обработчик сообщения WM_PAINT

```
   ...
case WM_PAINT:
     hdc = BeginPaint(hWnd, &ps);

     // Далее добавлен код для вывода строки сообщения
     // в окно приложения

     TextOut(hdc, 200, 100, textMes, lenText);
     EndPaint(hWnd, &ps);
     break;
   ...
```

Полный текст программы (назовем ее HELLOWC) на C++ с учетом сделанных изменений приведен в листинге 4.2.

Листинг 4.2. Программа, выводящая строку символов в окно приложения

```
// HELLOWC.cpp : точка входа нашего приложения

#include  "stdafx.h"
#include  "HELLOWC.h"
```

```
#define   MAX_LOADSTRING 100

// Объявление глобальных переменных

HINSTANCE hInst;                          // дескриптор экземпляра приложения
TCHAR     szTitle[MAX_LOADSTRING];            // заголовок окна приложения
TCHAR     szWindowClass[MAX_LOADSTRING];      // имя оконного класса

// Опережающие ссылки на функции, определенные в этом модуле

ATOM                 MyRegisterClass(HINSTANCE hInstance);
BOOL                 InitInstance(HINSTANCE, int);
LRESULT CALLBACK     WndProc(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
  // TODO: Place code here

  MSG msg;
  HACCEL hAccelTable;

  // Инициализация строковых ресурсов - можно пропустить
  // при первом чтении

  LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
  LoadString(hInstance, IDC_HELLOWC, szWindowClass, MAX_LOADSTRING);

  // Регистрация класса окна

  MyRegisterClass(hInstance);

  // Инициализация приложения

  if (!InitInstance(hInstance, nCmdShow))
  {
    return FALSE;
  }
```

```
  hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_HELLOWC);

  // Цикл обработки сообщений

  while (GetMessage(&msg, NULL, 0, 0))
  {
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return (int)msg.wParam;
}

// Функция MyRegisterClass(), выполняющая регистрацию
// класса окна приложения

ATOM MyRegisterClass(HINSTANCE hInstance)
{
  WNDCLASSEX wcex;
  wcex.cbSize        = sizeof(WNDCLASSEX);
  wcex.style         = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc   = (WNDPROC)WndProc;
  wcex.cbClsExtra    = 0;
  wcex.cbWndExtra    = 0;
  wcex.hInstance     = hInstance;
  wcex.hIcon         = LoadIcon(hInstance, (LPCTSTR)IDI_HELLOWC);
  wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
  wcex.lpszMenuName  = (LPCTSTR)IDC_HELLOWC;
  wcex.lpszClassName = szWindowClass;
  wcex.hIconSm       = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);

  return RegisterClassEx(&wcex);
}
```

```
// Функция InitInstance(HANDLE, int), выполняющая сохранение
// дескриптора экземпляра приложения и отображение главного
// окна приложения

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;

  hInst = hInstance; // сохраняем дескриптор приложения
                     // в глобальной переменной

  hWnd = CreateWindow(szWindowClass, szTitle,
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
                      CW_USEDEFAULT, 0, NULL, NULL,
                      hInstance, NULL);
  if (!hWnd)
  {
    return FALSE;
  }
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return TRUE;
}

// Функция WndProc(HWND, unsigned, WORD, LONG) выполняет обработку
// сообщений главного окна приложения. В этой реализации имеются два
// обработчика сообщений:
//    WM_PAINT - прорисовка главного окна
//    WM_DESTROY - уничтожение окна приложения

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
                         LPARAM lParam)
{
  PAINTSTRUCT ps;
  HDC hdc;

  // Эти переменные добавлены автором для демонстрации
  // вывода сообщения в окно приложения
```

```
  char *textMes = "HELLO FROM VISUAL C++ .NET !";
  int lenText = strlen(textMes);

  switch (message)
  {
    case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);

         // Здесь добавлен код для вывода строки сообщения
         // в окно приложения

         TextOut(hdc, 200, 100, textMes, lenText);
         EndPaint(hWnd, &ps);
         break;
    case WM_DESTROY:
         PostQuitMessage(0);
         break;
    default:
         return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

Проанализируем приведенный код. Начнем с идентификаторов, которые постоянно встречаются в исходном тексте программы. Они сведены в небольшую таблицу (табл. 4.1) в алфавитном порядке.

Таблица 4.1. *Идентификаторы, встречающиеся в листинге 4.2*

Идентификатор	Описание
CALLBACK	Используется вместо устаревшего соглашения Pascal в функции обратного вызова
HWND	Стандартное обозначение дескриптора окна
HANDLE	Дескриптор. Представляет собой 32-разрядное целое число без знака
HDC	Дескриптор контекста устройства
HINSTANCE	Дескриптор экземпляра приложения
LPARAM	Младший параметр сообщения (4 байта)

Таблица 4.1 (окончание)

Идентификатор	Описание
LPCSTR	32-разрядный указатель (адрес) строки-константы
LPSTR	32-разрядный указатель (адрес) строки
LPVOID	32-разрядный указатель общего типа
LRESULT	Используется для возврата значения из оконной процедуры
NULL	Общее обозначение нулевого значения
UINT	То же самое, что и int в C++ или Integer в Pascal
WCHAR	Представление символов в кодировке UNICODE (2 байта)
WINAPI	Используется вместо устаревшего соглашения Pascal при вызовах системных функций
WPARAM	Старший параметр сообщения (4 байта)

Из описания элементов таблицы видно, что многие типы идентификаторов представляют собой обычное двойное слово DWORD. Префикс LP некоторых идентификаторов указывает на то, что используется не значение переменной, а ее адрес.

Функция WinMain, являющаяся точкой входа в программу, определяется следующим образом:

```
int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
```

Для упрощения программирования с различными региональными настройками Microsoft разработала специфичные "общие" типы данных, процедур и других объектов. Такие типы данных в Visual C++ .NET объявляются с префиксом _t. В контексте данного примера объявление функции WinMain как _tWinMain принципиального значения не имеет.

WinMain использует вызовы функций WIN API и после завершения работы возвращает управление системе Windows. WinMain вызывается со следующими параметрами:

- hInstance. Этот параметр называется дескриптором (описателем) экземпляра приложения. Если выполняется несколько копий одной и той же программы, то каждая имеет свой уникальный дескриптор hInstance. Дескриптор представляет собой 32-разрядное двоичное число;

- hPrevInstance. Параметр считается устаревшим и всегда равен NULL;
- lpCmdLine. Этот параметр указывает на строку с завершающим нулем, в которой содержатся любые параметры, переданные в программу из командной строки;
- iCmdShow. Параметр указывает на вид окна в момент запуска приложения: SW_SHOWNORMAL (окно развернуто на экране) или SW_SHOWMINNOACTIVE (окно свернуто).

Главной задачей функции WinMain является создание окна приложения. Окно создается на основе класса окна. Используя один класс, можно создать несколько экземпляров окна. Кроме того, что класс окна определяет оконную процедуру, он устанавливает и другие характеристики окон, создаваемых на основе данного класса.

Ни одно оконное приложение не будет работать, если не выполнена регистрация класса окна. Поэтому перед созданием окна необходимо зарегистрировать класс окна путем вызова функции RegisterClassEx. В функцию RegisterClassEx передается один параметр: указатель на структуру типа WNDCLASSEX.

Программа WinMain должна вначале заполнить все поля структуры WNDCLASSEX определенными значениями, которые будут характеризовать наше окно — стиль, пиктограмма, цвет и др. В числе параметров должен быть указатель на функцию окна (оконную процедуру) нашей программы. Описание полей структуры WNDCLASSEX приведено в табл. 4.2.

Таблица 4.2. *Описание полей структуры* WNDCLASSEX

Описание поля	**Назначение**
UINT sbSize	Размер структуры в байтах
UINT style	Стиль окна. Можно использовать оператор ИЛИ (||) для комбинации стилей
WNDPROC lpfnWndProc	Указатель на оконную процедуру
int cbClsExtra	Информация о количестве байтов, выделенных операционной системой после того, как структура класса окна создана
int cbWndExtra	Информация о количестве байтов, выделенных операционной системой после создания экземпляра окна
HANDLE hInstance	Идентификатор экземпляра приложения, не может быть равен NULL
HICON hIcon	Определяет пиктограмму приложения, когда окно приложения свернуто

Таблица 4.2 (*окончание*)

Описание поля	Назначение
`HCURSOR hCursor`	Определяет, как будет выглядеть указатель мыши, если курсор будет находиться в рабочей области приложения
`HBRUSH hbrBackground`	Определяет тип кисти для заливки фоновой поверхности окна
`LPCTSTR lpszMenuName`	Указатель на строку с завершающим нулем, представляющую собой имя ресурса меню
`LPCTSTR lpszClassName`	Указатель на строку с завершающим нулем, представляющую собой имя класса окна
`HICON hIconSm`	Дескриптор пиктограммы, связанной с классом окна

Регистрация класса окна представлена следующим фрагментом программного кода из `WinMain` (листинг 4.3).

Листинг 4.3. Функция, выполняющая регистрацию класса окна

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{
  WNDCLASSEX wcex;
  wcex.cbSize        = sizeof(WNDCLASSEX);
  wcex.style         = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc   = (WNDPROC)WndProc;
  wcex.cbClsExtra    = 0;
  wcex.cbWndExtra    = 0;
  wcex.hInstance     = hInstance;
  wcex.hIcon         = LoadIcon(hInstance, (LPCTSTR)IDI_HELLOW);
  wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wcex.lpszMenuName  = (LPCTSTR)IDC_HELLOW;
  wcex.lpszClassName = szWindowClass;
  wcex.hIconSm       = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);
  return RegisterClassEx(&wcex);
}
```

Функция `MyRegisterClass` выполняет действия, необходимые для регистрации класса окна. Во-первых, в ней инициализируется структура

WNDCLASSEX. Во-вторых, вызывается функция RegisterClassEx, которая в качестве параметра принимает указатель на структуру WNDCLASSEX. Регистрация класса окна — это еще не само окно. Отображение окна и регистрация класса окна не имеют ничего общего. Оконный класс определяет общие характеристики окна и может использоваться для создания разных окон в одном приложении.

Само окно приложения создается в результате вызова функции CreateWindow. Эта функция детализирует информацию об окне. Разные окна одного и того же класса могут отличаться по размеру, располагаться в разных местах экрана, иметь отличные друг от друга заголовки. Эти характеристики и задаются при создании окна, а не класса окна.

Функция InitInstance отображает конкретное окно. Фрагмент программного кода представлен в листинге 4.4.

Листинг 4.4. Функция, выполняющая отображение окна приложения

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;
  hInst = hInstance;            // сохраняем дескриптор приложения
                                // в глобальной переменной
  hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                      NULL, NULL, hInstance, NULL);
  if (!hWnd)
  {
    return FALSE;
  }
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return TRUE;
}
```

Окно приложения создается функцией WIN API CreateWindow, имеющей синтаксис:

```
HWND CreateWindow(LPCTSTR lpClassName,     // имя класса окна
                  LPCTSTR lpWindowName,    // заголовок окна
                  DWORD   dwStyle,         // стиль окна
```

```
int      x,              // начальное положение окна
                         // по горизонтали
int      y,              // начальное положение окна
                         // по вертикали
int      nWidth,         // ширина окна
int      nHeight,        // высота окна
HWND     hWndParent,     // дескриптор родительского окна
HMENU    hMenu,          // дескриптор меню окна
HINSTANCE hInstance,     // дескриптор экземпляра
                         // приложения
LPVOID lpParam           // параметры создания окна
);
```

Третий параметр функции dwStyle указывает на стиль создаваемого окна. В нашей программе создается обычное перекрывающееся окно с заголовком, без меню, с пиктограммами для сворачивания, разворачивания и закрытия окна. Это стандартный стиль окон, который называется WS_OVERLAPPEDWINDOW. Параметры x и y задают начальные координаты верхнего левого угла окна относительно левого верхнего угла экрана. Если они равны CW_USEDEFAULT, то будет использоваться задаваемое по умолчанию начальное положение. Примерно так же задают ширину и высоту окна с помощью параметров nWidth и nHeight. CW_USEDEFAULT снова означает, что мы хотим, чтобы Windows использовала задаваемый по умолчанию размер окна.

Чтобы отобразить окно на экране, необходимо вызвать функцию ShowWindow. В эту функцию передаются два параметра — hWnd и nCmdShow. Параметр hWnd — это дескриптор окна, возвращенный функцией CreateWindow. Параметр nCmdShow указывает, как должно отображаться окно при первом появлении на экране. Если присвоить этому параметру значение SW_SHOWNORMAL, то окно будет развернуто на экране. Если параметру присвоить значение SW_SHOWMINIMIZED, то окно появится в свернутом виде.

И, наконец, функция UpdateWindow генерирует сообщение WM_PAINT, указывающее на необходимость прорисовки рабочей области окна. После регистрации класса окна и отображения экземпляра окна на экране функция WinMain переходит к бесконечному циклу обработки сообщений. Исходный текст этого цикла приведен в листинге 4.5.

Листинг 4.5. Цикл обработки сообщений

```
while (GetMessage(&msg, NULL, 0, 0))
{
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
```

```
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

В цикле обработки сообщений используется обычный оператор цикла `while`. Как мы знаем, Windows посылает каждому приложению сообщение о каком-либо событии, на которое приложение должно соответствующим образом реагировать. Любое сообщение из очереди сообщений приложения можно получить при помощи функции `GetMessage`. Функция просматривает очередь сообщений приложения и копирует выбранное сообщение в структуру, адрес которой `&msg` является ее первым параметром. В этой структуре имеются два поля — `LPARAM` и `WPARAM`, в которые записывается код полученного сообщения.

Второй параметр функции указывает на окно приложения, которому направлено сообщение. Если он равен `NULL`, то выбираются сообщения, направленные всем окнам приложения. Два последних параметра помогают сформировать фильтр для сообщений. Если оба они равны 0, то приложение пропускает все сообщения.

В цикле `while` используются еще несколько функций. Функция `TranslateMessage` преобразует сообщения о нажатии виртуальных клавиш в символьные сообщения и удобна, если приложение активно работает с клавиатурой. Наконец, функция `DispatchMessage` посылает сообщение, сохраненное в `&msg`, соответствующей оконной процедуре приложения.

Приложение будет находиться в цикле обработки сообщений до тех пор, пока не получит сообщение `WM_QUIT`. В этом случае функция `GetMessage` возвращает `FALSE`, и происходит выход из программы. Далее рассмотрим оконную процедуру `WndProc`, исходный текст которой приведен в листинге 4.6.

Листинг 4.6. Оконная процедура WndProc

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
                         LPARAM lParam)
{
  PAINTSTRUCT ps;
  HDC hdc;

  char *textMes = "HELLO FROM VISUAL C++ .NET!";
  int lenText = strlen(textMes);
```

```
  switch (message)
  {
    case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);
         TextOut(hdc, 200, 100, textMes, lenText);
         EndPaint(hWnd, &ps);
         break;
    case WM_DESTROY:
         PostQuitMessage(0);
         break;
   default:
         return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

В качестве параметров оконная процедура принимает:

- дескриптор `hWnd` окна, которому операционная система Windows посылает сообщение;
- идентификатор сообщения `message`, которое должно быть обработано;
- `WPARAM` и `LPARAM`, которые хранят дополнительную информацию о сообщении.

В самой оконной процедуре определены переменные `hdc` и `ps`. Переменная `hdc` содержит дескриптор контекста устройства отображения, а переменная `ps` определяет структуру `PAINTSTRUCT`, в которой хранится информация о параметрах прорисовки окна.

Более подробно контекст и элементы структуры `PAINTSTRUCT` будут рассмотрены в *главе 5*.

Окно работающего приложения, скомпилированного в Visual C++ .NET, изображено на рис. 4.1.

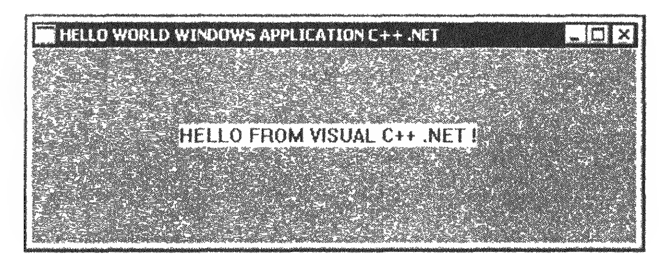


Рис. 4.1. Окно стандартного приложения Windows, разработанного на C++ .NET

Теперь посмотрим, как будет выглядеть подобная программа на языке ассемблера, выводящая строку "ПРИВЕТ ИЗ АССЕМБЛЕРА!" в окно приложения.

Было бы очень удобно воспользоваться каркасом приложения на C++ и сделать нечто подобное на ассемблере. Посмотрим, что для этого необходимо. Наше приложение использует в основном функции WIN API. Язык ассемблера довольно легко позволяет манипулировать с функциями WIN API с помощью команды `call`. Если функциям API требуется передавать параметры, то это делается через стек. Покажем, как изменить программный код C++ .NET в ассемблерный на примере функции `TextOut`.

Вот интересующий нас фрагмент кода на C++:

```
  ...
char *textMes = "HELLO FROM VISUAL C++ .NET !";
int lenText = strlen(textMes);
  ...
TextOut(hdc, 200, 100, textMes, lenText);
  ...
```

В программе на ассемблере подобный фрагмент с соответствующими изменениями текста сообщения может быть представлен следующим образом:

```
   ...
.data
   ...
  textMes      DB  "ПРИВЕТ ИЗ АССЕМБЛЕРА!"
  lenText      EQU $-textMes
   ...
.code
   ...
  push    lenText
  push    offset textMes
  push    100
  push    100
  push    hdc
  call    TextOut
   ...
```

Цикл `while` обработки сообщений, написанный на C++, также реализуется при помощи ассемблерных команд сравнения и условных переходов. Программный код цикла на C++ представлен в листинге 4.7.

Листинг 4.7. Цикл обработки сообщений на C++

```
while (GetMessage(&msg, NULL, 0, 0))
{
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

В том виде, в котором он представлен, цикл обработки сообщений слишком сложен, поэтому первое, что мы сделаем, — упростим его. Мы не используем в нашем приложении "быстрые" клавиши (акселераторы), поэтому обработчик сообщений для таких клавиш будет отсутствовать. Соответственно, не нужно проверять условие:

```
if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
```

Уберем этот условный оператор, и фрагмент кода упростится (листинг 4.8).

Листинг 4.8. Упрощенный вариант цикла обработки сообщений

```
while (GetMessage(&msg, NULL, 0, 0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

Проанализировав упрощенный цикл обработки сообщений, не составит особых усилий написать аналогичный цикл на языке ассемблера (листинг 4.9).

Листинг 4.9. Цикл обработки сообщений на языке ассемблера

```
StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
```

```
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop
  lea     EAX, msg
  push    EAX
  call    TranslateMessage
  lea     EAX, msg
  push    EAX
  call    DispatchMessage
  jmp     StartLoop

ExitLoop:
  mov     EAX, msg.wParam
  ret
```

Наконец, представим исходный текст оконной процедуры `WndProc` на ассемблере. Напомню, что исходный текст оконной процедуры `WndProc` на C++ был приведен ранее в листинге 4.6. Не составит труда, учитывая наш предыдущий опыт, написать эквивалент оконной процедуры на ассемблере (листинг 4.10).

Листинг 4.10. Оконная процедура `WndProc` на языке ассемблера

```
WndProc proc hWin    :DWORD,
             uMsg    :DWORD,
             wParam  :DWORD,
             lParam  :DWORD

  LOCAL hdc   :HDC
  LOCAL ps    :PAINTSTRUCT

  cmp     uMsg, WM_PAINT
  jne     next_1

  lea     EDX, ps
  push    EDX
  push    hWnd
```

```
  call    BeginPaint
  mov     hdc, EAX

  push    lenText
  push    offset textMes
  push    100
  push    100
  push    hdc
  call    TextOut

  lea     EDX, ps
  push    EDX
  push    hWnd
  call    EndPaint

  ret
next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX

  ret
next_2:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc

  ret
WndProc endp
```

Мы проанализировали, как работает классическое Windows-приложение, написанное на C++, и знаем, как реализовать основные функциональные блоки такой программы на ассемблере. Прежде чем представить наше первое графическое приложение на языке ассемблера, вспомним, как выглядит в общих чертах каркас такого приложения. Для разработки приложений

Windows мы будем использовать макроассемблер MASM. Структура приложения на ассемблере представлена в листинге 4.11.

Листинг 4.11. Структура приложения на языке ассемблера

```
.386
.model flat, stdcall
.data
  <данные>
.code
метка:
  <программный код>
end <метка>
```

Большинство выражений и директив ассемблера, которые используются в этом шаблоне, уже встречались в предыдущих главах. Кратко напомним их назначение.

В первой строке находится директива `.386`, которая указывает ассемблеру на то, что будет использоваться набор команд 386-го процессора. Директива `.model flat` указывает ассемблеру, что используется плоская модель памяти.

Как мы уже знаем из предыдущих глав, директива `stdcall` указывает ассемблеру на порядок передачи параметров при вызове внешних процедур. Параметры в процедуру передаются справа налево, причем первый параметр помещается в стек первым. Вызываемая процедура также должна восстановить стек. Директива `stdcall` используется потому, что при разработке Windows-приложений на ассемблере мы в основном будем использовать WIN API, а почти все функции (за исключением `wsprintf`) этого интерфейса передают параметры в соответствии с этой директивой. К примеру, если мы используем функцию WIN API с условным названием `my_func` с четырьмя параметрами `param1`, `param2`, `param3`, `param4`, определенную как `my_func(param1, param2, param3, param4)`, то для вызова ее из программы на ассемблере необходимо выполнить следующую последовательность команд:

```
push param4
push param3
push param2
push param1
call my_func
```

В принципе в приложениях на ассемблере можно использовать любые внешние процедуры, например из библиотек Visual Studio .NET. Эти процедуры могут иметь другие соглашения о вызовах, например cdecl.

Директива .data определяет область данных для программы. Это означает, что в непрерывном адресном пространстве выделяется логический сегмент данных. Если в 16-разрядных приложениях для инициализации области данных программист обычно использовал регистр DS, то в 32-разрядных приложениях об этом заботиться не нужно.

Директива .code обозначает начало кода нашей программы.

Теперь мы знаем достаточно, чтобы приступить к разработке первого приложения на ассемблере в операционной системе Windows. Программа будет выводить в окно текст "ПРИВЕТ ИЗ АССЕМБЛЕРА!". Исходный текст программы (назовем ее HELLOW) приведен в листинге 4.12.

Листинг 4.12. Программа HELLOW, выводящая в окно строку символов

```
;----------------- HELLOW.ASM -------------------

.386
.model flat, stdcall
  option casemap : none                        ; различаем регистр символов
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;------ Прототипы функций -----------

  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB  "ПЕРВОЕ ГРАФИЧЕСКОЕ ПРИЛОЖЕНИЕ НА АССЕМБЛЕРЕ", 0
  CommandLine   DD  0
  hWnd          DD  0
```

```
  hInstance     DD  0
  szClassName   DB  "Demo_Class", 0
  textMes       DB  "ПРИВЕТ ИЗ АССЕМБЛЕРА!"
  lenText       EQU $-textMes

.code

start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX

  call    GetCommandLine
  mov     CommandLine, EAX

  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain

  push   EAX
  call   ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc    :WNDCLASSEX
  LOCAL msg   :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами

  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW
  ;
```

```
mov     wc.lpfnWndProc, offset WndProc
mov     wc.cbClsExtra, NULL
mov     wc.cbWndExtra, NULL
;
push    hInst
pop     wc.hInstance
;
mov     wc.hbrBackground, COLOR_BTNFACE+1
mov     wc.lpszMenuName, NULL
mov     wc.lpszClassName, offset szClassName
;
push    IDI_APPLICATION
push    NULL
call    LoadIcon
mov     wc.hIcon, EAX

push    IDC_ARROW
push    NULL
call    LoadCursor
mov     wc.hCursor, EAX
mov     wc.hIconSm, 0
;
lea     EAX, wc
push    EAX
call    RegisterClassEx
;
push    NULL
push    hInst
push    NULL
push    NULL
;
push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT
;
push    WS_OVERLAPPEDWINDOW
push    offset szDisplayName
```

```
  push    offset szClassName
  push    WS_EX_OVERLAPPEDWINDOW
  call    CreateWindowEx
  mov     hWnd, EAX

  push    SW_SHOWNORMAL
  push    hWnd
  call    ShowWindow

  push    hWnd
  call    UpdateWindow

  ; Здесь выполняется цикл обработки сообщений

StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop

  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage

  jmp     StartLoop
ExitLoop:

  mov     EAX, msg.wParam
  ret
WinMain endp
```

```
; Оконная процедура нашего приложения

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc  :HDC
  LOCAL ps   :PAINTSTRUCT

  cmp     uMsg, WM_PAINT
  jne     next_1
  lea     EDX, ps
  push    EDX
  push    hWnd
  call    BeginPaint
  mov     hdc, EAX

  push    lenText
  push    offset textMes
  push    100
  push    100
  push    hdc
  call    TextOut

  lea     EDX, ps
  push    EDX
  push    hWnd
  call    EndPaint
  ret
next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret
```

```
next_2:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret
WndProc endp

end start
```

Для компиляции и сборки такой программы при помощи макроассемблера MASM необходимо выполнить в командной строке следующую последовательность команд:

```
ml /c /coff hellow.asm
link /SUBSYSTEM:WINDOWS /LIBPATH: <disk>:\masm\lib hellow.obj
```

Опция `/c` компилятора `ml` указывает на то, что нужно создать только объектный модуль без вызова компоновщика. Опция `/coff` предписывает компилятору создать объектный файл в формате `COFF`. Компоновщик использует опцию `/SUBSYSTEM:WINDOWS` для генерации 32-разрядного Windows-приложения. Опция `/LIBPATH` указывает компоновщику местонахождение библиотек импорта.

Мы видим, что в исходном тексте появились новые строки:

```
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\gdi32.inc

includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
```

Директивы `includelib` указывают компилятору на библиотеки импорта. Директивы `include` подключают к программе файлы с расширением INC, в которых содержится важная информация. В windows.inc определены константы и структуры, используемые в программировании 32-разрядных

приложений. Файлы kernel32.inc и user32.inc содержат записи о прототипах функций из системных библиотек kernel32.dll и user32.dll соответственно. Строка записи в таком файле имеет вид:

```
Имя_Функции PROTO [имя_парам_1]:тип, [имя_парам_2]:тип ...
```

где:

- `Имя_Функции` — идентификатор процедуры;
- `[имя_парам]` — необязательное имя параметра;
- `тип` — тип параметра.

Например, прототипы функций `CreateWindow` и `UpdateWindow` описаны так:

```
ShowWindow PROTO :DWORD,:DWORD
UpdateWindow PROTO :DWORD
```

Каждому прототипу соответствует определенная функция в библиотеке динамической компоновки. Но возникают вопросы: зачем нужны прототипы, и что дает нам их использование? Еще раз внимательно посмотрим на листинг исходного кода. Для вызова, например, функции `ShowWindow` требуется выполнить последовательность команд:

```
push    SW_SHOWNORMAL
push    hWnd
call    ShowWindow
```

Прототипы функций позволяют нам использовать так называемый высокоуровневый вызов процедур и функций при помощи выражения (оператора) `invoke`:

```
invoke <имя_функции или указатель>, [аргументы]
```

В этом случае вызов функции `ShowWindow` можно представить так:

```
invoke  ShowWindow, hWnd, SW_SHOWNORMAL
```

Приведем еще один пример. Вызов функции `TextOut` можно представить при помощи оператора `invoke` следующим образом:

```
invoke  TextOut, hdc, 100, 100, ADDR textMes, lenText
```

Оператор ADDR в выражении invoke для функции TextOut используется для передачи адреса переменной. Этот оператор работает только вместе с оператором invoke и самостоятельно не применяется. Однако вы можете использовать и ключевое слово offset для получения адреса переменной при работе с invoke, например:

```
invoke  TextOut, hdc, 100, 100, offset textMes, lenText
```

Надо отметить, что есть одно маленькое неудобство в применении оператора ADDR: он не работает, если в программе имеется опережающая ссылка, т. е. если выражение invoke встретится в программе раньше, чем ссылка на переменную. В этом случае компилятор MASM выдаст ошибку. С оператором offset такого не бывает.

Остальная часть исходного текста программы состоит из рассмотренных ранее фрагментов кода и в дополнительных комментариях не нуждается. Поскольку мы довольно часто будем применять оператор invoke, то предлагается еще один вариант программы HELLOW, где большинство вызовов функций WIN API выполняется именно с помощью этого оператора (листинг 4.13).

Листинг 4.13. Вариант программы HELLOW, использующий оператор invoke

```
;----------------- HELLOW.ASM (вариант 2)--------------

.386
.model flat, stdcall
  option casemap :none                          ; различаем регистр букв

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;------ Прототипы функций ------------

  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
```

```
.data
  szDisplayName DB  "ПЕРВОЕ ГРАФИЧЕСКОЕ ПРИЛОЖЕНИЕ НА АССЕМБЛЕРЕ", 0
  CommandLine   DD  0
  hWnd          DD  0
  hInstance     DD  0
  szClassName   DB  "Demo_Class", 0
  textMes       DB  "ПРИВЕТ ИЗ АССЕМБЛЕРА!"
  lenText       EQU $-textMes
.code

start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX
  invoke  GetCommandLine
  mov     CommandLine, EAX
  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, 0

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc   :WNDCLASSEX
  LOCAL msg  :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами

  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW
  ;
  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL
  ;
  push    hInst
  pop     wc.hInstance
```

```
  ;
  mov     wc.hbrBackground, COLOR_BTNFACE+1
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName

  invoke  LoadIcon, NULL, IDI_APPLICATION
  mov     wc.hIcon, EAX
  invoke  LoadCursor, NULL, IDC_ARROW

  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0
  invoke  RegisterClassEx, ADDR wc
  ;
  invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, _USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL
  mov     hWnd,EAX
  invoke  ShowWindow, hWnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hWnd

  ; Здесь выполняется цикл обработки сообщений

StartLoop:
  invoke  GetMessage, ADDR msg, NULL, 0, 0
  cmp     EAX, 0
  je      ExitLoop
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

; Оконная процедура нашего приложения

WndProc proc hWin    :DWORD,
             uMsg    :DWORD,
```

```
              wParam :DWORD,
              lParam :DWORD

   LOCAL hdc  :HDC
   LOCAL ps   :PAINTSTRUCT

   cmp     uMsg, WM_PAINT
   jne     next_1
   invoke  BeginPaint, hWnd, ADDR ps
   mov     hdc, EAX
   invoke  TextOut, hdc, 100, 100, ADDR textMes, lenText
   invoke  EndPaint, hWnd, ADDR ps
   ret
next_1:
   cmp     uMsg, WM_DESTROY
   jne     next_2
   invoke  PostQuitMessage, NULL
   xor     EAX, EAX
   ret
next_2:
   invoke  DefWindowProc, hWin, uMsg, wParam, lParam
   ret
WndProc endp
end start
```

Окно работающего приложения изображено на рис. 4.2.



Рис. 4.2. Окно стандартного приложения Windows на ассемблере

Для всех примеров следующих глав используются одни и те же параметры командной строки компилятора MASM. Полное описание директив компилятора не приводится, чтобы не загромождать текст излишними деталями и подробностями, которые нам вряд ли понадобятся. Существует много хороших описаний компилятора MASM, в которых эти директивы подробно описаны, и читатель при желании сможет найти любую интересующую его информацию.

Мы не будем использовать макросредства и высокоуровневые структуры языка ассемблер (кроме оператора `invoke`) — они упрощают исходный текст, но затрудняют анализ программ. Наши приложения должны быть легко читаемы и анализируемы! Исходные тексты программ этой и следующей глав легко адаптируются для работы с компилятором Borland TASM 5.

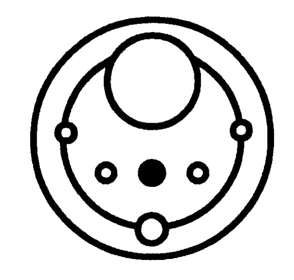
Можно сделать некоторые выводы о программировании Windows-приложений на ассемблере:

- во-первых, многие приложения, подчас довольно сложные, можно написать, используя шаблон классического приложения, модифицируя существующие обработчики сообщений в оконной процедуре и/или применяя свои;
- во-вторых, богатый набор WIN API функций позволяет решить практически любую задачу по обработке данных;
- в-третьих, использование ассемблера существенно увеличивает быстродействие программы и уменьшает размер исполняемого модуля.

Можно надеятся, что материал последующих глав еще больше убедит читателя использовать язык ассемблера для разработки Windows-приложений.



Глава 5



Программирование на ассемблере в Windows: от простого к сложному

В предыдущей главе мы разработали простую программу для Windows, выводящую в окно приложения строку символов. В этой главе мы последовательно рассмотрим основные аспекты эффективного программирования на ассемблере в операционной среде Windows. Эффективно программировать на ассемблере в Windows можно только в том случае, если научиться гармонично сочетать преимущества языка низкого уровня и возможности самой операционной системы. Можно с уверенностью утверждать, что качество программы во многом зависит от того, насколько хорошо программист владеет знаниями архитектуры Windows и умением использовать функции прикладного интерфейса программирования (WIN API). В этой главе, как и в остальных, сделан основной упор на практический аспект программирования. Мы будем рассматривать практически полезные программы и фрагменты кода.

Экзотические программы, такие как вывод текста вертикально или конструирование необычных геометрических фигур, мы рассматривать не будем. Для совершенствования техники программирования они могут представлять какой-то интерес, но для практического применения в разработках бесполезны.

Поскольку большинство приложений работает с графическим интерфейсом, то необходимо вначале понять, как операционная система манипулирует графическими объектами, как осуществляется программирование графической подсистемы Windows.

GDI (Graphics Devise Interface — графический интерфейс устройства) — подсистема Windows, отвечающая за отображение текста и графики на экранах и принтерах. GDI является важнейшей компонентой операционной системы.

Графический интерфейс используется не только нашими приложениями, но и сама система Windows активно использует его для отображения элементов пользовательского интерфейса, таких как меню, полосы прокрутки, значки,

курсоры мыши и другие графические объекты. Графика в 32-битной Windows реализуется в основном функциями, экспортируемыми из динамической библиотеки GDI32.DLL.

Для программиста графический интерфейс представляет собой набор функций и нескольких связанных с ними типов данных, макросов и структур. Но прежде чем рассмотреть некоторые из этих функций подробно, остановимся на общей структуре GDI.

5.1. Графический интерфейс Windows

Все функции графического интерфейса можно разбить на несколько групп. Это весьма условное деление, поскольку операции, выполняемые разными функциями, могут частично перекрываться. Для нас будут иметь важное значение следующие группы функций:

1. функции, создающие или уничтожающие контекст устройства. К ним относятся GetDC, ReleaseDC, BeginPaint и EndPaint. Все эти функции позволяют получить дескриптор контекста устройства отображения. Последние две функции используются обычно внутри обработчика WM_PAINT для рисования. Функции GetDC и ReleaseDC позволяют получить дескриптор в обработчиках сообщений, отличных от WM_PAINT;
2. функции, которые получают информацию о контексте устройства. К таким функциям относится, например, GetTextMetrics, возвращающая информацию о выбранном шрифте. Другая функция GetDeviceCaps позволяет получить информацию об указанном устройстве (дисплее или принтере). Еще одна функция, GetGraphicsMode, возвращает информацию о текущем графическом режиме. Кроме этих, в Windows имеется огромное количество других функций, несущих информацию о тех или иных системных установках;
3. функции, выполняющие вывод графических объектов в окно приложения. Например, для вывода текста используются функции DrawText и TextOut. Также имеется множество других функций, позволяющих рисовать линии и растровые изображения.

Рассмотрим более подробно понятие "контекста устройства". Контекст устройства (Device Context — DC) — это структура данных, которая поддерживается GDI. Контекст связан с конкретным устройством вывода информации, таким как принтер или дисплей. Почему мы должны уделить столь пристальное внимание контексту устройства? Дело в том, что эта структура используется при выполнении всех операций ввода-вывода информации на дисплей, принтер и, возможно, другие устройства.

Для дисплея, например, контекст устройства связан с окном приложения. Определенные значения в контексте устройства являются графическими

атрибутами, определяющими параметры функций рисования. Например, функция TextOut использует такие атрибуты, как цвет текста, цвет фона для текста и шрифт, используемый для вывода текста.

Контекст устройства содержит много атрибутов, определяющих работу функций графического интерфейса с устройством. Обычно для работы функций GDI требуются лишь начальные координаты или размеры. Например, если мы используем функцию TextOut, то необходимо в ее параметрах указать только дескриптор (описатель) контекста устройства, начальные координаты, сам выводимый текст и его длину. Дескриптор контекста передается обычно в переменной hdc, которая определяется в оконной процедуре так:

```
HDC  hdc
```

Данные типа HDC представляют собой 32-разрядное целое беззнаковое число. После получения дескриптора контекста программа может использовать функции графического интерфейса, такие как TextOut и DrawText.

Указывать шрифт, цвет текста, цвет фона и расстояние между отдельными символами не нужно, поскольку эти атрибуты являются частью контекста устройства. Если по каким-либо причинам необходимо изменить один из этих атрибутов, то нужно вызвать соответствующую функцию. Перед началом рисования программа должна получить дескриптор контекста устройства. По окончании рисования программа должна освободить дескриптор. После освобождения дескриптор становится недействительным и использоваться не должен. Корректно работающая программа должна получать и освобождать дескриптор во время обработки каждого отдельного сообщения.

Дескриптор контекста устройства может быть получен одним из двух способов:

- при обработке сообщений WM_PAINT. В этом случае используются функции BeginPaint и EndPaint. В качестве параметров эти функции принимают дескриптор окна и адрес структуры PAINTSTRUCT, обычно именуемой как ps. Оконная процедура вызывает функцию BeginPaint в обработчике сообщения WM_PAINT. Функция возвращает в качестве результата дескриптор контекста в переменной типа HDC, именуемой обычно hdc. После получения дескриптора контекста можно использовать функции рисования, например, TextOut или DrawText. Вызов функции EndPaint освобождает дескриптор контекста устройства. Процесс обработки сообщения WM_PAINT, например в программе HELLOW из *главе 4*, будет выглядеть так, как представлено в листинге 5.1.

Листинг 5.1. Обработка сообщения WM_PAINT

```
  ...
cmp     uMsg, WM_PAINT
jne     next_1

lea     EDX, ps
push    EDX
push    hWnd
call    BeginPaint
mov     hdc, EAX

push    lenText
push    offset textMes
push    100
push    100
push    hdc
call    TextOut
lea     EDX, ps
push    EDX
push    hWnd
call    EndPaint
ret
  ...
```

В случае, если сообщение WM_PAINT не обрабатывается, то оно должно передаваться в процедуру обработки сообщений по умолчанию DefWindowProc. Процедура DefWindowProc обрабатывает сообщения WM_PAINT, вызвав подряд функции BeginPaint и EndPaint, как показано в следующем фрагменте кода:

```
  ...
lea     EDX, ps
push    EDX
push    hWnd
call    BeginPaint
mov     hdc, EAX

lea     EDX, ps
```

```
push    EDX
push    hWnd
call    EndPaint
ret
  ...
```

- при обработке сообщений, отличных от `WM_PAINT`, если возникает необходимость рисования, вызвав функцию `GetDC`. Освободить дескриптор контекста можно с помощью функции WIN API `ReleaseDC`. Функция `GetDC` в качестве параметра принимает дескриптор окна приложения. Функция `ReleaseDC` в качестве параметров принимает дескриптор окна и дескриптор контекста, ранее созданный через вызов `GetDC`.

Вывод текста и работу с контекстом устройства рисования с помощью функций WIN API `BeginPaint` и `EndPaint` мы рассмотрели в *главе 4*. Поэтому проанализируем второй метод получения дескриптора контекста. Для этого модифицируем наше приложение `HELLOW`.

Пусть требуется вывести строку текста при нажатии в окне приложения левой кнопки мыши. В этом случае в оконную процедуру необходимо включить обработчик сообщения `WM_LBUTTONDOWN`. Исходный текст программы (назовем ее `DRAWTEXT`) приведен в листинге 5.2.

Листинг 5.2. Исходный текст программы DRAWTEXT

```
;------------------- DRAWTEXT.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none                          ; различаем регистр символов

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------        szDisplayName  DB  "КОНТЕКСТ УСТРОЙСТВА И ВЫВОД ТЕКСТА
(ВАРИАНТ 2)", 0
  CommandLine    DD  0
                                          .code
```

```
start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX
  invoke  GetCommandLine
  mov     CommandLine, EAX
  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, 0

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
   ; Локальные переменные процедуры
   LOCAL wc   :WNDCLASSEX
  LOCAL msg :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами

  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style,  CS_HREDRAW or CS_VREDRAW
  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra,  NULL
  mov     wc.cbWndExtra,  NULL
  push    hInst
  pop     wc.hInstance
   mov     wc.hbrBackground, COLOR_BTNFACE+1
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName
  invoke   , NULL, IDI_APPLICATION   mov     wc.hIcon, EAX

  invoke   , NULL, IDC_ARROW   mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0
  invoke  RegisterClassEx, ADDR wc   invoke  CreateWindowEx,
WS_EX_OVERLAPPEDWINDOW,ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL
      invoke  ShowWindow, hWnd, SW_SHOWNORMAL   invoke  UpdateWindow,
hWnd   ; Цикл обработки сообщений
```

```
StartLoop:
  invoke  GetMessage, ADDR msg, NULL, 0, 0
  cmp     EAX, 0
  je      ExitLoop
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc   :HDC
  LOCAL rect  :RECT
  LOCAL coord :DWORD

  cmp     uMsg, WM_LBUTTONDOWN
  jne     next_1
  invoke  GetDC, hWnd
  mov     hdc, EAX
  invoke  GetClientRect, hWnd, ADDR rect
  mov     coord, DT_CENTER or DT_SINGLELINE or DT_VCENTER
  invoke  DrawText, hdc, ADDR textDraw, -1, ADDR rect, coord
  invoke  ReleaseDC, hWnd, hdc
  ret
next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
  invoke  PostQuitMessage, NULL
  xor     EAX, EAX
  ret
next_2:
  invoke  DefWindowProc,hWin, uMsg, wParam, lParam
  ret
```

```
WndProc endp
end start
```

Для читателей, предпочитающих классический ассемблер без высокоуровневых структур, таких как `if-else`, `while` и `invoke`; приводим исходный текст только что рассмотренного примера. В листинге 5.3 приведен текст программы, где используются только команды ассемблера.

Листинг 5.3. Программа DRAWTEXT, в которой используются только команды ассемблера

```
;------------ DRAWTEXT.ASM (классический стиль) ---------
.386
.model flat, stdcall
  option casemap :none                        ; различаем регистр символов

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------    WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB  "КОНТЕКСТ УСТРОЙСТВА И ВЫВОД ТЕКСТА (ВАРИАНТ 2)", 0
  CommandLine   DD  0
  hWnd          DD  0
  hInstance     DD  0
  szClassName   DB  "Demo_Class", 0
  textDraw      DB  " ", 0
  lenText       EQU $-textDraw
.code

start:
  push    NULL
```

```
  call    GetModuleHandle
  mov     hInstance, EAX
  call    GetCommandLine
  mov     CommandLine, EAX

  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain
  push    EAX
  call    ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
   ; Локальные переменные процедуры
   LOCAL wc    :WNDCLASSEX
  LOCAL msg   :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами
   mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW
  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL

  push    hInst
  pop     wc.hInstance
  mov     wc.hbrBackground, COLOR_BTNFACE+1
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName
   push    IDI_APPLICATION
  push    NULL
  call    LoadIcon
  mov     wc.hIcon, EAX
```

```
  push    IDC_ARROW
  push    NULL
  call    LoadCursor
  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0

  lea     EAX, wc
  push    EAX
  call    RegisterClassEx   push    NULL
  push    hInst
  push    NULL
  push    NULL

  push    CW_USEDEFAULT
  push    CW_USEDEFAULT
  push    CW_USEDEFAULT
  push    CW_USEDEFAULT

  push    WS_OVERLAPPEDWINDOW
  push    offset szDisplayName
  push    offset szClassName
  push    WS_EX_OVERLAPPEDWINDOW
  call    CreateWindowEx
  mov     hWnd,EAX
  push    SW_SHOWNORMAL
  push    hWnd
  call    ShowWindow
  push    hWnd
  call    UpdateWindow
  ; Цикл обработки сообщений
StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage
  cmp     EAX, 0
```

```
  je      ExitLoop
  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage

  jmp     StartLoop
ExitLoop:
       WinMain endp

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
   LOCAL hdc    :HDC
  LOCAL ps     :PAINTSTRUCT
  LOCAL rect   :RECT
  LOCAL coord :DWORD                     ; используется для хранения
                                         ; параметров форматирования

  ; Обработчик нажатия левой кнопки мыши

  cmp     DWORD PTR [EBP+12], WM_LBUTTONDOWN   jne     next_1
  push    hWnd
  call    GetDC
  mov     hdc, EAX

  lea     ESI,rect
       push    hWnd
  call    GetClientRect

  mov     coord, DT_CENTER or DT_SINGLELINE or DT_VCENTER
  push    coord
  lea     ESI, rect
  push    ESI
  push    -1
```

```
  push    offset textDraw
  push    hdc
  call    DrawText
  push    hdc
  push    hWnd
  call    ReleaseDC
  ret
next_1:
  cmp     DWORD PTR [EBP+12], WM_DESTROY   jne    next_2
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret
next_2:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret
WndProc endp
```

Как видно из исходного текста, рисование текста выполняется без использования обработчика сообщения `WM_PAINT`, и дескриптор контекста устройства мы получаем с помощью функции `GetDC` в обработчике нажатия левой кнопки мыши `WM_LBUTTONDOWN`. После вывода текста в окно приложения контекст устройства освобождается функцией `ReleaseDC`.

В этом примере в качестве вспомогательной мы использовали функцию `GetClientRect`. Эта функция возвращает координаты клиентской области окна в структуре `RECT`, которую мы назовем `rect`. Структура имеет вид:

```
struct {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT;
```

Поля структуры имеют следующее назначение:

- `left` определяет горизонтальную координату `x` левого верхнего угла клиентской области окна;
- `top` определяет вертикальную координату `y` левого верхнего угла клиентской области;
- `right` определяет координату `x` правого нижнего угла клиентской области окна;
- `bottom` определяет координату `y` правого нижнего угла клиентской области окна.

Координаты клиентской области окна вычисляются относительно левого верхнего угла (0, 0). Значения полей структуры `rect` используются функцией `DrawText` для вывода текста. Эта функция рисует отформатированный текст в определенной области окна и имеет следующий синтаксис:

```
int DrawText(HDC      hdc,          // дескриптор контекста
             LPCTSTR  lpString,     // указатель на выводимую строку
             int      nCount,       // размер строки
             LPRECT   lpRect,       // указатель на структуру RECT
             UINT     uFormat       // опции форматирования
            );
```

Если параметр `nCount` равен `-1`, то предполагается, что `lpString` является указателем на строку с завершающим нулем, и функция `DrawText` вычисляет размер строки автоматически. Наконец, последний параметр `uFormat` устанавливает опции форматирования текста. В нашем случае текст располагается в одну строку по середине клиентской области окна.

Окно работающего приложения изображено на рис. 5.1.



Рис. 5.1. Окно приложения, отображающего текст с помощью функции `DrawText`

Во всех последующих примерах и фрагментах кода будем сочетать использование высокоуровневых операторов invoke с обычными командами ассемблера. Это поможет избежать как громоздкости кода, так и его малой информативности.

5.2. Вывод текста на экран: дополнительные возможности

Для вывода текста на экран монитора во многих случаях удобно использовать функцию WIN API TextOut, с которой мы уже встречались. В последующих примерах для отображения текста мы будем применять в основном ее.

Как вы уже поняли, отобразить текст в окне приложения нетрудно. Но если требуется выравнивание строк текста по горизонтали или вывод в строго определенные позиции окна, то программист сталкивается с некоторыми сложностями. Следует учитывать и то, что при изменении размеров окна относительное расположение текста не меняется. Это приводит к тому, что видимые части текста могут просто исчезать.

Многие программы нуждаются в позиционировании текста определенным образом. Функции DrawText, TextOut и другие имеют весьма ограниченные возможности по форматированию и позиционированию текста.

К счастью, в Windows есть целый ряд функций, при помощи которых можно добиться очень точного расположения текста в окне. Далее мы разработаем программу, при помощи которой можно отображать текст посередине клиентской области окна.

Вначале немного теории. Функция TextOut в качестве параметров, помимо всего прочего, принимает горизонтальное и вертикальное смещения начальной точки рисования относительно левого верхнего угла окна. Задать координаты можно прямо в операторе, например:

```
TextOut(hdc, 100, 50, lpString, lenString),
```

где lpString — адрес строки, lenString — ее размер.

В этом случае точка начала рисования текста будет отстоять на 100 единиц по горизонтали и на 50 по вертикали от начала отсчета. Но какие единицы измерения используются для рисования в Windows и для каких систем координат? Здесь необходимо дать некоторые пояснения.

Координаты графического интерфейса в документации Microsoft упоминаются как "логические координаты" (logical coordinates). Окно описывается в терминах логических координат. Ими могут быть пикселы, миллиметры,

дюймы или любые другие единицы, какие мы захотим. В вызовах функций GDI мы задаем логические координаты.

В Windows имеются различные режимы отображения (mapping mode), которые определяют, как логические координаты, заданные в функциях GDI, преобразуются в реальные физические координаты дисплея. Режим отображения определяется в контексте устройства. Задаваемый по умолчанию режим отображения называется `MM_TEXT` (идентификатор, заданный в заголовочных файлах Windows).

В этом режиме отображения логические единицы (logical units) эквивалентны физическим единицам, что позволяет нам работать непосредственно в терминах пикселов. В дальнейшем при анализе примеров программ будут использоваться только логические единицы.

Вернемся к функции `TextOut`. Задавать координаты в виде чисел не совсем удобно, поэтому чаще применяют другой способ позиционирования текста. Для этого нужно вычислить координаты клиентской части окна приложения при помощи функции `GetClientRect`. Мы уже знаем, как работает эта функция, и сейчас отобразим строку текста посередине клиентской области. В этом случае программный код будет выглядеть так, как представлено в листинге 5.4.

Листинг 5.4. Отображение текста посередине клиентской области окна приложения

```
   ...
.data
  textOut DB  "Text"
  lenText EQU $-textOut
   ...
  LOCAL   hdc  :HDC
  LOCAL   rect :RECT
 LOCAL   x, y :DWORD   горизонтальная и вертикальная координаты
   ...
                        invoke  GetClientRect, hWnd, ADDR rect
   ...
  push    EBX
  mov     EBX, rect.right
  sub     EBX, rect.left
  mov     x, EBX

  mov     EBX, rect.bottom
```

```
sub     EBX, rect.top
mov     y, EBX
pop     EBX
 ...
invoke  TextOut, hdc, x, y, ADDR textOut, lenText
 ...
```

Как видно из исходного текста, координаты x и y будут точно указывать на середину клиентской области окна. Однако текст не будет располагаться симметрично относительно точек x и y, поскольку в нашем коде не учтен тот факт, что строка имеет определенную длину. Здесь возникает вопрос: каким образом подсчитать размер строки в логических единицах, чтобы правильно расположить текст в окне? Подобная задача довольно часто встречается в программистской практике.

Логично было бы предположить, что размер строки равен приблизительно горизонтальному размеру (ширине) символа, умноженному на количество символов в строке. Остается определить ширину символа. Ширину символа в логических единицах можно получить несколькими способами. Рассмотрим их по порядку.

Первый способ — с помощью функции GetTextMetrics. Функция имеет синтаксис:

```
BOOL GetTextMetrics(HDC hdc, LPTEXTMETRIC lptm)
```

В качестве параметров функция принимает дескриптор контекста устройства (hdc) и указатель (lptm) на структуру TEXTMETRIC, содержащую информацию о выбранном шрифте.

После вызова функции можно проанализировать значения полей в структуре TEXTMETRIC и сохранить некоторые из них для дальнейшего использования.

Особенно интересными для нас являются два поля: tmAveCharWidth и tmHeight. Первое поле tmAveCharWidth определяет среднюю ширину шрифта, а второе tmHeight — его высоту. Этих данных вполне достаточно для того, чтобы написать программу, отображающую строку текста посередине окна и симметрично относительно его центра. Поскольку размеры системного шрифта не меняются в рамках одного сеанса работы с Windows, достаточно вызвать функцию GetTextMetrics только один раз при выполнении программы.

Исходный текст программы (назовем ее OUTTM) представлен в листинге 5.5.

Листинг 5.5. Вывод строки посередине окна приложения с помощью функции GetTextMetrics

```
;---------------------- OUTTM.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------     WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD   WndProc PROTO
:DWORD,:DWORD,:DWORD,:DWORD
 .data
  szDisplayName DB  "ПОЗИЦИОНИРОВАНИЕ ТЕКСТА С ИСПОЛЬЗОВАНИЕМ
GetTextMetric", 0
  CommandLine   DD  0
  hWnd          DD  0
  hInstance     DD  0   szClassName   DB  "Demo_Class", 0  textOut
DB  "Текст отображается функцией TextOut"
 lenText       EQU $-textOut
.code
start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX
  invoke  GetCommandLine

  mov     CommandLine, EAX
  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, 0
WinMain proc hInst     :DWORD,              hPrevInst :DWORD,
             CmdLine   :DWORD,
```

```
            CmdShow    :DWORD

  ; Локальные переменные процедуры
   LOCAL wc    :WNDCLASSEX
       :

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами

  mov     wc.cbSize, sizeof WNDCLASSEX  mov     wc.style, CS_HREDRAW or
CS_VREDRAW
 mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL

  push    hInst
  pop     wc.hInstance

  mov     wc.hbrBackground, COLOR_BTNFACE+9
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName

  invoke  LoadIcon, NULL, IDI_APPLICATION
  mov     wc.hIcon, EAX

  invoke  LoadCursor, NULL, IDC_ARROW
  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0
  invoke  RegisterClassEx, ADDR wc
  invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL
  mov     hWnd,EAX
  invoke  ShowWindow,hWnd,SW_SHOWNORMAL
  invoke  UpdateWindow,hWnd

; Цикл обработки сообщений
StartLoop:
  invoke  GetMessage, ADDR msg, NULL, 0, 0
```

```
  cmp     EAX, 0
  je      ExitLoop
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc    :HDC
  LOCAL rect   :RECT
  LOCAL tm     :TEXTMETRIC
  LOCAL tx, ty :DWORD
  LOCAL x, y   :DWORD

  cmp     uMsg, WM_LBUTTONDOWN
  jne     next_1
  invoke  GetDC, hWnd
  mov     hdc, EAX
  invoke  GetClientRect, hWnd, ADDR rect

  ; Позиционируем текст по горизонтали. Для этого получаем
  ; параметры шрифта и сохраняем их в переменных tx и ty

  invoke  GetTextMetrics, hdc, ADDR tm
  mov     EAX, tm.tmAveCharWidth
  mov     tx, EAX
  mov     EAX, tm.tmHeight
  mov     ty, EAX

  push    EBX
  mov     EBX, rect.right
  sub     EBX, rect.left
```

```
  ; Вычисляем размер строки как произведение ширины символа(tx)
  ; на количество символов в строке (lenText)+1

  mov     EAX, tx
  mov     ECX, lenText
  inc     ECX
  mul     ECX
   ; Вычисляем начальную точку вывода текста по горизонтали
   sub     EBX, EAX
  shr     EBX, 1
  mov     x, EBX

  ; Позиционируем строку текста по вертикали
  ; с учетом высоты строки (ty)

  mov     EAX, rect.bottom
  sub     EAX, rect.top
  sub     EAX, ty
  shr     EAX, 1
  mov     y, EAX   invoke  TextOut, hdc, x, y, ADDR textOut, lenText
  invoke ReleaseDC, hWnd, hdc
  ret
next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
  invoke  PostQuitMessage, NULL
  xor     EAX, EAX
  ret
next_2:
  invoke  DefWindowProc, hWin, uMsg, wParam, lParam
  ret
WndProc endp
end start
```

Необходимо заметить, что параметры шрифта во многом зависят от характеристик дисплея, поэтому лучше не задавать никаких фиксированных зна-

чений, а использовать функцию GetTextMetrics. Окно работающего приложения изображено на рис. 5.2.

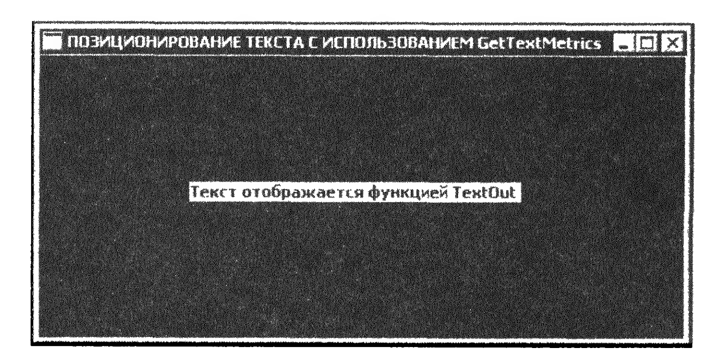


Рис. 5.2. Окно приложения, отображающего позиционирование текста с использованием GetTextMetrics

Второй способ точного позиционирования текста — использование функции GetTextExtentPoint32. Эта функция WIN API выполняет подсчет ширины и высоты строк в логических единицах. Функция имеет синтаксис:

```
BOOL GetTextExtentPoint32(HDC     hdc,        // дескриптор контекста
                          LPCTSTR lpString,   // указатель на строку
                          int     cbString,   // количество символов
                                              // в строке lpString
                          LPSIZE  lpSize  //указатель на структуру SIZE
                         );
```

Важное замечание: строка символов lpString не обязательно должна завершаться нулем, т. к. размер строки все равно определен в параметре cbString.

Параметр lpSize указывает на структуру типа SIZE, определяющую ширину и высоту прямоугольника, ограничивающего строку. Структура может быть представлена так:

```
struct tagSIZE {
  LONG  cx;
  LONG  cy;
} SIZE, *PSIZE;
```

Поля cx и cy определяют, соответственно, ширину и высоту прямоугольника.

Исходный текст программы (назовем ее TEXTP) представлен в листинге 5.6.

Листинг 5.6. Вывод текста посередине окна приложения с помощью функции `GetTextExtentPoint32`

```
;---------------------- TEXTP.ASM -----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------
.data
  szDisplayName DB  "Позиционирование текста с помощью GetTextExtent-
Point32", 0
  CommandLine   DD  0
  hWnd          DD  0
  hInstance     DD  0
  szClassName   DB  "Demo_Class", 0
  textOut       DB  "Текст отображается функцией TextOut"
  lenText       EQU $-textOut
  tsize         label DWORD
  crx           DD  0
  cry      DD  0 .code

start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX

  invoke  GetCommandLine
  mov     CommandLine, EAX
  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, 0
```

```
WinMain proc hInst     :DWORD,
                    CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc  :WNDCLASSEX
  LOCAL msg :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами
   mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW
   ov     wc.lpfnWndProc, offset WndProc
          wc.cbClsExtra, NULL
          wc.cbWndExtra, NULL

  push    hInst
  pop     wc.hInstance   mov     wc.hbrBackground, COLOR_BTNFACE+1
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName

  invoke  LoadIcon, NULL, IDI_APPLICATION
  mov     wc.hIcon, EAX
   invoke  LoadCursor, NULL, IDC_ARROW
  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0

  invoke  RegisterClassEx, ADDR wc
  invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL
  mov     hWnd, EAX
  invoke  ShowWindow, hWnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hWnd
 ;Цикл обработки сообщений
 StartLoop:
  invoke  GetMessage, ADDR msg, NULL, 0,
```

```
  cmp     EAX, 0
  je      ExitLoop
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc   :HDC
  LOCAL rect  :RECT
  LOCAL tm    :TEXTMETRIC
  LOCAL x, y  :DWORD

  cmp     uMsg, WM_LBUTTONDOWN
  jne     next_1
  invoke  GetDC, hWnd
  mov     hdc, EAX
  invoke  GetClientRect, hWnd, ADDR rect invoke  GetTextExtentPoint32,
hdc, ADDR textOut, lenText, ADDR tsize
  mov     EAX, rect.right
  sub     EAX, rect.left
  sub     EAX, crx
  shr     EAX, 1
  mov     x, EAX

  mov     EAX, rect.bottom
  sub     EAX, rect.top
  sub     EAX, cry
  shr     EAX, 1
  mov     y, EAX

  invoke  TextOut, hdc, x, y, ADDR textOut, lenText
```

```
  invoke  ReleaseDC, hWnd, hdc
  ret
next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
  invoke  PostQuitMessage, NULL
  xor     EAX, EAX
  ret
next_2:
  invoke DefWindowProc,hWin, uMsg, wParam, lParam
  ret
WndProc endp end start
```

Проанализируем исходный текст программы. Чтобы воспользоваться функцией `GetTextExtentPoint32`, в секции `.data` определим аналог структуры `SIZE`:.

```
   ...
.data
   ...
  tsize   label DWORD
  crx     DD 0
  cry   DD 0    ...
```

Переменные `crx` и `cry` хранят размеры прямоугольника текста (ширину и высоту).

Чтобы получить значения переменных, в исходном тексте программы, точнее, в обработчике нажатия левой кнопки мыши `WM_LBUTTONDOWN`, должны присутствовать следующие строки:

```
  ...
invoke  GetTextExtentPoint32, hdc, ADDR textOut, lenText, ADDR tsize
mov     EAX, rect.right
sub     EAX, rect.left
sub     EAX, crx
 shr     EAX, 1 mov     x, EAX
mov     EAX, rect.bottom
sub     EAX, rect.top
```

```
sub     EAX, cry
shr     EAX, 1
mov     y, EAX
invoke  TextOut, hdc, x, y, ADDR textOut, lenText
 ...
```

Окно работающего приложения изображено на рис. 5.3.

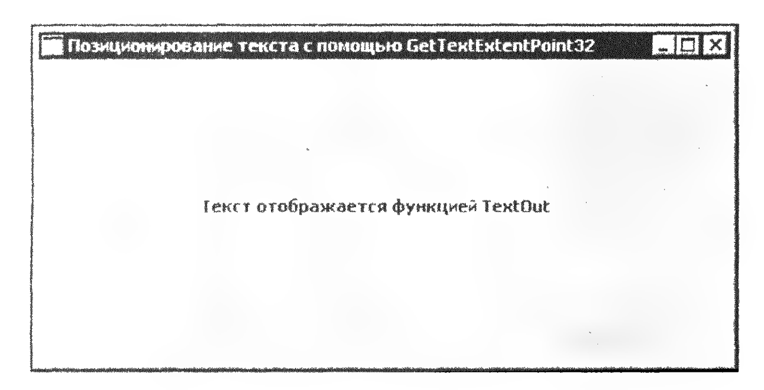


Рис. 5.3. Окно приложения, отображающего позиционирование текста с помощью GetTextExtentPoint32

5.3. Работа со шрифтами

До сих пор мы выводили текст, используя стандартные установки операционной системы для цвета и размера шрифта. При написании графических приложений редко случается так, что программист обходится только системными шрифтами. Далее мы разработаем программу, в которой продемонстрируем выбор шрифта с нужными характеристиками для вывода строки текста. В качестве шаблона выберем приложение, выводящее несколько строк различными шрифтами в клиентскую область окна. Прежде всего определим, какой функцией воспользоваться для установки атрибутов шрифта. В Windows есть функция `CreateFont` для инициализации логических шрифтов с заданными характеристиками. Логический шрифт заменяет шрифт по умолчанию для любого устройства. Сама функция имеет синтаксис:

```
HFONT CreateFont(int    nHeight,        // высота шрифта
                 int    nWidth,         // средняя ширина шрифта
                 int    nEscapement,
```

```
                  int     nOrientation,
                  int     fnWeight,            // "жирность" шрифта
                  DWORD   fdwItalic,
                  DWORD   fdwUnderline,
                  DWORD   fdwStrikeOut,
                  DWORD   fdwCharSet,          // кодировка символов
                                               // (ANSI или UNICODE)
                  DWORD   fdwOutputPrecision,
                  DWORD   fdwClipPrecision,
                  DWORD   fdwQuality,
                  DWORD   fdwPitchandFamily,
                  LPCTSTR lpszFace);   // адрес строки с названием шрифта
```

В определении функции дана расшифровка тех параметров, которые наиболее существенны и используются программистами наиболее часто.

Разработаем приложение, в котором при каждом нажатии на правую кнопку мыши шрифт строки текста "`Текст отображается функцией TextOut`" будет увеличиваться и становиться более жирным, а при нажатии на левую кнопку, наоборот, уменьшаться. Текст будет отображаться при помощи функции `TextOut` посередине клиентской области окна.

В нашей оконной процедуре `WndProc` будут присутствовать обработчики сообщений `WM_PAINT`, `WM_LBUTTONDOWN` и `WM_RBUTTONDOWN`. Исходный текст программы (назовем ее `SELFONT`) приведен в листинге 5.7.

Листинг 5.7. Программа SELFONT, демонстрирующая установку аттрибутов шрифта

```
;------------------- SELFONT.ASM -----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib
```

```
;--------    WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
 WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
 szDisplayName DB  "ВЫБОР ШРИФТОВ С ПОМОЩЬЮ SelectFont", 0
 CommandLine   DD  0
 hWnd          DD  0
 hInstance     DD  0
 szClassName   DB  "Demo_Class", 0
 textOut       DB  "Текст отображается функцией TextOut"
 lenText       EQU $-textOut

 tsize         label DWORD
 crx           DD  0
 cry           DD  0

 myf           DB  "Arial Cyr", 0
 mypitch       EQU DEFAULT_PITCH or FF_SWISS
 myq           EQU DEFAULT_QUALITY
 myclip        EQU CLIP_DEFAULT_PRECIS
 myout         EQU OUT_DEFAULT_PRECIS
 myansi        EQU ANSI_CHARSET
 vHeight       DD  0
.code
start:
 invoke  GetModuleHandle, NULL
 mov     hInstance, EAX

 invoke  GetCommandLine
 mov     CommandLine, EAX
 mov     vHeight, 14
 invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
 invoke  ExitProcess, 0
WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
```

```
    ; Локальные переменные процедуры

   LOCAL wc  :WNDCLASSEX
   LOCAL msg :MSG

   ; Заполнение структуры WNDCLASSEX требуемыми параметрами

   mov     wc.cbSize, sizeof WNDCLASSEX
   mov     wc.style, CS_HREDRAW or CS_VREDRAW

   mov     wc.lpfnWndProc, offset WndProc
   mov     wc.cbClsExtra, NULL
   mov     wc.cbWndExtra, NULL
    push    hInst
   pop     wc.hInstance

   mov     wc.hbrBackground, COLOR_BTNFACE+2
   mov     wc.lpszMenuName, NULL
   mov     wc.lpszClassName, offset szClassName

   invoke  LoadIcon, NULL, IDI_APPLICATION
   mov     wc.hIcon, EAX
    invoke  LoadCursor, NULL, IDC_ARROW
   mov     wc.hCursor, EAX
   mov     wc.hIconSm, 0
   invoke  RegisterClassEx, ADDR wc
   invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                       ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                       CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                       CW_USEDEFAULT, NULL, NULL, hInst, NULL
   mov     hWnd, EAX
   invoke  ShowWindow,hWnd,SW_SHOWNORMAL
   invoke  UpdateWindow,hWnd

; Цикл обработки сообщений
StartLoop:
   invoke GetMessage, ADDR msg, NULL, 0, 0
   cmp     EAX, 0
```

```
   je       ExitLoop
   invoke   TranslateMessage, ADDR msg
   invoke   DispatchMessage, ADDR msg
   jmp      StartLoop
ExitLoop:
   mov      EAX, msg.wParam
   ret
WinMain endp

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

   ; Локальные переменные

   LOCAL hdc     :HDC
   LOCAL ps      :PAINTSTRUCT
   LOCAL rect    :RECT

   LOCAL myFont  :HFONT
   LOCAL tx, ty  :DWORD
   LOCAL x, y    :DWORD

   cmp      uMsg, WM_LBUTTONDOWN
   jne      next_1
   invoke   GetClientRect, hWin, ADDR rect
   cmp      vHeight, 14
   jge      down
   mov      vHeight, 25
   jmp      wtext_1
down:
   dec      vHeight
wtext_1:
   invoke   InvalidateRect, hWin, ADDR rect, TRUE
   ret
```

```
next_1:
  cmp     uMsg, WM_RBUTTONDOWN
  jne     next_2
  invoke  GetClientRect, hWin, ADDR rect
  cmp     vHeight, 25
  jge     minHeight
  inc     vHeight
  jmp     wtext_2
minHeight:
  mov     vHeight, 14
wtext_2:
       ret
next_2:
  cmp     uMsg, WM_PAINT
  jne     next_3
  lea     EDX, ps
  push    EDX
  push    hWin
  call    BeginPaint
  mov     hdc, EAX
   invoke  GetClientRect, hWin, ADDR rect
                 mov     myFont, EAX
  invoke  SelectObject, hdc,myFont

  push    myFont
  push    hdc
  call    SelectObject

  invoke  GetTextExtentPoint32, hdc, ADDR textOut, lenText, ADDR tsize
  mov     EAX, rect.right
  sub     EAX, rect.left
  sub     EAX, crx
  shr     EAX, 1
  mov     x, EAX

  mov     EAX, rect.bottom
  sub     EAX, rect.top
  sub     EAX, cry
  shr     EAX, 1
```

```
  mov     y, EAX
  invoke  TextOut, hdc, x, y, ADDR textOut, lenText

  lea     EDX, ps
  push    EDX
  push    hWin
  call    EndPaint   ret

next_3:
  cmp     uMsg, WM_DESTROY
  jne     next_4
  invoke  PostQuitMessage, NULL
  xor     EAX, EAX
  ret

next_4:
  invoke  DefWindowProc, hWin, uMsg, wParam, lParam
  ret
WndProc endp
end start
```

Приложение работает следующим образом: каждый раз при нажатии левой кнопки мыши размер шрифта уменьшается на 1, а при нажатии на правую кнопку мыши — увеличивается на 1. Соответственно будет меняться и вид отображаемой в окне строки текста. Минимальное значение высоты шрифта установлено равным 14 единицам, максимальное значение — 25. Шрифт выбран Arial Cyr нормальной толщины.

В этой программе, кроме демонстрации вывода текста шрифтами различных размеров, показан вариант обработки сообщений WM_PAINT. Это очень важный момент, и поэтому акцентируем на нем особое внимание. Обработчик сообщения WM_PAINT оконной процедуры отвечает за правильное отображение информации в окне приложения. Хорошо спроектированная оконная процедура в любой момент времени должна отображать истинную информацию, будь-то текст или другой графический образ на экране. Это означает, что при любом изменении положения окна, его размеров, после закрытия ниспадающего меню или диалогового окна, перекрывавшего окно приложения, информация должна быть немедленно восстановлена.

При запуске приложения функция UpdateWindow генерирует первое сообщение WM_PAINT, вынуждающее программу перерисовать клиентскую об-

ласть окна. Это же сообщение посылается приложению и в ряде других случаев:

- при вызове функций `InvalidateRect` или `InvalidateRgn`;
- при изменении размеров окна;
- при вызове функции `ScrollWindow`;
- после закрытия диалогового окна, перекрывавшего окно нашего приложения.

Очень удобно вывод данных выполнить в обработчике сообщения `WM_PAINT`, а предварительную обработку данных — в обработчиках других сообщений. Это можно сделать, если, например, в обработчиках нажатия кнопок мыши последней вызывать функцию `InvalidateRect`. Вызов `InvalidateRect` вынуждает операционную систему Windows пометить клиентскую область окна приложения как недействительную и послать окну приложения сообщение `WM_PAINT`. Последней в обработчике сообщения `WM_PAINT` вызывается функция `EndPaint`, которая сообщает Windows, что перерисовка окна закончена.

Вновь вернемся к нашей программе. В оконной процедуре используется три обработчика сообщений: `WM_PAINT`, `WM_LBUTTONDOWN`, `WM_RBUTTONDOWN`.

Размер шрифта устанавливается в переменной `vHeight`. При нажатии левой кнопки мыши размер шрифта будет уменьшаться, при нажатии правой кнопки, наоборот, увеличиваться. Исходный текст обработчика сообщения `WM_LBUTTONDOWN` оконной процедуры будет выглядеть следующим образом:

```
   ...
  cmp     uMsg, WM_LBUTTONDOWN
  jne     next_1
  invoke  GetClientRect, hWin, ADDR rect
  cmp     vHeight, 14
  jge     down

  mov     vHeight, 25
  jmp     wtext_1
down:
  dec     vHeight
wtext_1:
  invoke  InvalidateRect, hWin, ADDR rect, TRUE
  ret
   ...
```

Обратите внимание на то, что в обработчике необходимо вызвать функцию `GetClientRect` для инициализации струтуры `rect`. Если этого не сделать, то последующий вызов функции `InvalidateRect` ни к чему не приведет! Как видно из исходного текста обработчика, размер шрифта при каждом щелчке левой кнопки мыши уменьшается на 1 в диапазоне от 25 до 14. Вызов функции `InvalidateRect` вынуждает Windows генерировать сообщение `WM_PAINT`, и обработчик этого сообщения прорисовывает текст с новым размером шрифта.

Обработчик нажатия правой кнопки мыши работает аналогично обработчику левой, с той лишь разницей, что размер шрифта увеличивается от 14 до 25. Операции позиционирования текста и отображения его в окне приложения выполняются в обработчике сообщения `WM_PAINT`. Назначение контексту устройства нового шрифта для вывода текста выполняется в следующем фрагменте программного кода:

```
  ...
invoke CreateFont, vHeight, 0, 0, 0, 400, 0, 0, 0, myansi, myout,\
                   myclip, myq, mypitch, ADDR myf
mov     myFont, EAX
invoke  SelectObject, hdc,myFont
  ...
```

Назначение и принцип работы функции `CreateFont` нам известны, а вот зачем нужна функция `SelectObject`? Эта функция устанавливает вновь созданный объект в контекст устройства, заменяя ранее установленный. Этим объектом может быть шрифт, кисть или битовый образ. Функция имеет следующий синтаксис:

```
HGDIOBJ SelectObject(HDC      hdc,        // дескриптор контекста
                     HGDIOBJ  hgdiobj);   // дескриптор объекта
```

Позиционирование текста посередине клиентской области выполняется при помощи уже знакомой нам функции `GetTextExtentPoint32`.

Для читателей, предпочитающих "чистый" ассемблер, приводится реализация функций `CreateFont` и `SelectObject`:

```
push    offset myf
push    mypitch
push    myq push   myclip
push   myout

push    myansi
```

```
push    0
push     0 push    0

push     400
push     0
push     0
push     0

push     vHeight call    CreateFont mov     myFont, EAX
push    myFont
push    hdc
call     SelectObject
```

Вид окна работающего приложения при последовательных щелчках правой кнопкой мыши изображен на рис. 5.4 и 5.5.

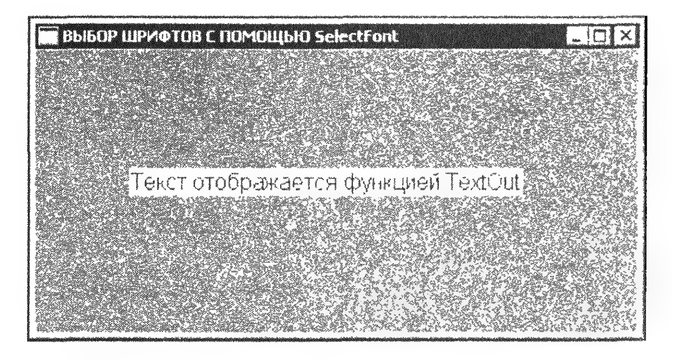


Рис. 5.4. Окно приложения, устанавливающего переменный размер шрифта при 6-м щелчке правой кнопкой мыши

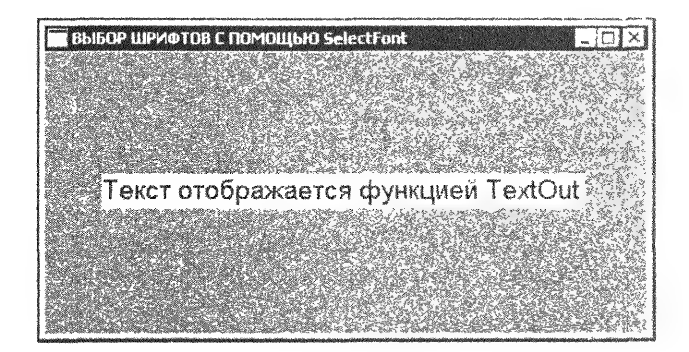


Рис. 5.5. Окно приложения, устанавливающего переменный размер шрифта при 7-м щелчке правой кнопкой мыши

Рассмотрим еще один пример работы с текстом. Попробуем решить такую задачу: вывести три строки текста в клиентскую область окна, причем размеры шрифта для каждой строки будут отличаться. Строки необходимо отобразить с определенным интервалом в клиентской области окна приложения. Все эти операции выполним в обработчиках сообщений WM_LBUTTONDOWN, WM_RBUTTONDOWN и WM_PAINT оконной процедуры. Исходный текст программы представлен в листинге 5.8.

Листинг 5.8. Программа, выводящая в клиентскую область окна три строки с разными размерами шрифта

```
;--------------------- SSTR_1.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------    WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName  DB  "ВЫВОД НЕСКОЛЬКИХ СТРОК ТЕКСТА РАЗНЫМИ ШРИФТАМИ", 0
  CommandLine    DD  0
  hWnd           DD  0
  hInstance      DD  0
  szClassName    DB  "Demo_Class", 0

  myStr          label BYTE
  s1             DB  "СТРОКА 1", 0
  s2             DB  "СТРОКА 2", 0
  s3             DB  "СТРОКА 3", 0
  ls             DD  0
```

```
  stepy          DD  5
  stepx          DD  40

  myf            DB  "Arial Cyr", 0
        myq              EQU DEFAULT_QUALITY
  myclip         EQU CLIP_DEFAULT_PRECIS
  myout          EQU OUT_DEFAULT_PRECIS
  myansi         EQU ANSI_CHARSET

  vHeight        DD  0

.code

start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX
  call    GetCommandLine
  mov     CommandLine, EAX
  mov     vHeight, 12

  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain
  push    EAX
  call    ExitProcess
 WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  LOCAL wc  :WNDCLASSEX
  LOCAL msg :MSG

  ;  Заполнение структуры WNDCLASSEX

  mov     wc.cbSize, sizeof WNDCLASSEX
```

```
mov     wc.style, CS_HREDRAW or CS_VREDRAW

mov     wc.lpfnWndProc, offset WndProc
mov     wc.cbClsExtra, NULL
mov     wc.cbWndExtra, NULL

push    hInst
pop     wc.hInstance
 mov     wc.hbrBackground, COLOR_BTNFACE+1
mov     wc.lpszMenuName, NULL
mov     wc.lpszClassName, offset szClassName
 push    IDI_APPLICATION
push    NULL
call    LoadIcon
mov     wc.hIcon, EAX

push    IDC_ARROW
push    NULL
call    LoadCursor
mov     wc.hCursor, EAX
mov     wc.hIconSm, 0
        push    EAX
call    RegisterClassEx

push    NULL
push    hInst
push    NULL
push    NULL
 push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT

push    WS_OVERLAPPEDWINDOW
push    offset szDisplayName
push    offset szClassName
push    WS_EX_OVERLAPPEDWINDOW
call    CreateWindowEx
mov     hWnd, EAX
```

```
  push    SW_SHOWNORMAL
  push    hWnd
  call    ShowWindow

  push    hWnd
  call    UpdateWindow
 ; Цикл обработки сообщений
StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop
  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
  LOCAL hdc      :HDC
  LOCAL ps       :PAINTSTRUCT
  LOCAL rect     :RECT
  LOCAL myFont   :HFONT
```

```
LOCAL tm      :TEXTMETRIC
LOCAL ty      :DWORD
LOCAL cnt     :DWORD
LOCAL x, y    :DWORD
 cmp     uMsg, WM_PAINT
jne     next_1
lea     EDX, ps
push    EDX
push    hWin
call    BeginPaint
mov     hdc, EAX
 invoke  GetClientRect, hWin, ADDR rect
invoke  GetTextMetrics, hdc, ADDR tm
mov     EAX, tm.tmHeight
shl     EAX, 1
mov     ty, EAX
 mov     EAX, rect.right
sub     EAX, rect.left
xor     EDX, EDX
mov     ECX, 3
div     ECX
mov     x, EAX

mov     EAX, rect.bottom
sub     EAX, rect.top
xor     EDX, EDX
mov     ECX, 5
div     ECX
add     EAX, stepy
mov     y, EAX
 push    vHeight
mov     cnt, 1
lea     ESI, myStr

next_raw:
  invoke  CreateFont, vHeight, 0, 0, 0, 600, 0, 0, 0, myansi, myout,\
                      myclip, myq, mypitch, ADDR myf
  mov     myFont, EAX
```

```
  invoke  SelectObject, hdc,myFont

  push    ESI
  call    LenStr
  mov     ls, EAX   push    ls
  push    ESI

  push    y
  push    x
  push    hdc
  call    TextOut
  add     vHeight, 3
  mov     EAX, x
  add     EAX, stepx
  mov     x, EAX

  mov     EAX, ty
 add     y, EAX
  add     ESI, ls
  inc     ESI
  cmp     cnt, 3
  je      cont
  inc     cnt
  mp      next_raw
cont:
  lea     EDX, ps
  push    EDX
  push    hWin
  call    EndPaint
  pop     vHeight
  ret

next_1:
  cmp     uMsg, WM_LBUTTONDOWN
  jne     next_2
  invoke  GetClientRect, hWin, ADDR rect
  add     stepy, 5
  invoke  InvalidateRect, hWin, ADDR rect, TRUE
  ret
```

```
next_2:
  cmp     uMsg, WM_RBUTTONDOWN
  jne     next_3
  invoke  GetClientRect, hWin, ADDR rect
  sub     stepy, 5
  invoke  InvalidateRect, hWin, ADDR rect, TRUE
  ret

next_3:
           4   push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret

next_4:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret
WndProc endp

LenStr proc
  push    EBP
  mov     EBP, ESP
  mov     EDX, [EBP+8]
  mov     EDI, EDX
  cld
  mov     AL, 0
next_check:
  scasb
  je      ex
  jmp     next_check
ex:
  sub     EDI, EDX
  mov     EAX, EDI
```

```
  dec     EAX
  pop     EBP
  ret     4
LenStr endp
end start
```

Окно работающего приложения изображено на рис. 5.6.

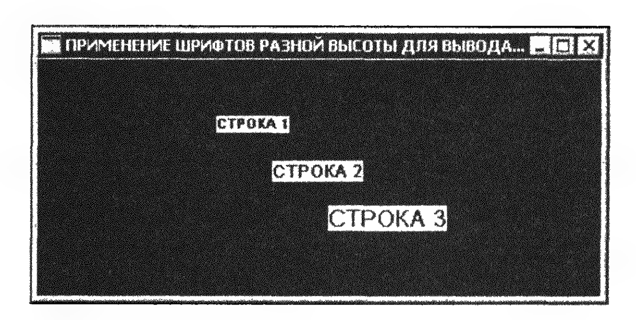


Рис. 5.6. Окно приложения, в котором выполняется вывод трех строк текста разными шрифтами

Вывод текста в клиентскую область окна выполняется в обработчике WM_PAINT с помощью функции TextOut. Строки выводимого текста находятся по вертикали на расстоянии, определяемом переменной ty, которая вычисляется как удвоенная высота шрифта, полученного при помощи функции GetTextMetrics:

```
  ...
invoke  GetTextMetrics, hdc, ADDR tm
mov     EAX, tm.tmHeight
shl     EAX, 1
mov     ty, EAX
  ...
```

Для каждой из строк вычисляется свой размер шрифта (переменная vHeight). Для каждой следующей строки высота увеличивается на 3. В обработчике WM_PAINT установка шрифта выполняется следующим фрагментом кода:

```
   ...
next_raw:
  invoke  CreateFont, vHeight, 0, 0, 0, 600, 0, 0, 0, myansi, myout,\
                      myclip, myq, mypitch, ADDR myf
```

```
mov     myFont, EAX
invoke  SelectObject, hdc,myFont
 ...
```

Кроме высоты шрифта, для каждой строки определяется смещение по горизонтали (координата x) и по вертикали (координата y). Это осуществляется двумя фрагментами кода в обработчике WM_PAINT:

```
 ...
add     vHeight, 3
mov     EAX, x
add     EAX, stepx
mov     x, EAX
 ...
mov     EAX, ty
add     y, EAX
 ...
```

При выводе каждой последующей строки к координате x добавляется смещение, заданное в переменной stepx. Координата y также меняется:

```
 ...
add     EAX, stepy
mov     y, EAX
 ...
```

В обработчиках нажатия кнопок мыши выполняется приращение переменных stepx и stepy, причем приращение stepy в обработчике WM_LBUTTONDOWN имеет положительный знак (строки текста сдвигаются вниз), а в обработчике WM_RBUTTONDOWN — отрицательный (строки текста сдвигаются вверх). Визуально это отображается синхронным перемещением всех трех строк по вертикали.

Для вывода строки текста на экран необходимо знать ее размер. Он вычисляется при помощи процедуры LenStr, принимающей в качестве единственного параметра адрес строки с завершающим нулем. Исходный текст процедуры понятен, поэтому останавливаться на его анализе я не буду.

До сих пор мы рассматривали геометрические параметры шрифтов. Для многих программ этого, однако, недостаточно. Помимо таких атрибутов как высота шрифта, его тип, толщина, хотелось бы устанавливать и его цветовые параметры. В операционной системе Windows имеется несколько функ-

ций для установки цветовых атрибутов шрифта. Мы воспользуемся одной из таких функций — `SetTextColor`. При помощи этой функции можно установить цвет шрифта, определенного в контексте устройства рисования. Функция имеет синтаксис:

```
COLORREF SetTextColor(HDC       hdc,          // дескриптор контекста
                      COLORREF  crColor);     // цвет текста
```

Первый параметр функции — дескриптор контекста устройства, а о втором параметре следует рассказать более подробно. Переменная `COLORREF` может быть представлена в 16-ричном формате как `0x00bbggrr`, где младший байт содержит величину, характеризующую относительную интенсивность красного цвета, второй байт определяет относительную интенсивность зеленого, а старший байт характеризует интенсивность синего цвета. Самый старший байт должен равняться 0. Максимальная величина для одного байта равна `0xFF`.

Модифицируем только что рассмотренный пример для вывода строк текста на экран разными шрифтами. Будем по-прежнему отображать те же три строки текста, но разными цветами. Цвет шрифта будет меняться при нажатии правой кнопки мыши в окне приложения, а также при изменении размеров окна.

Кроме демонстрации работы с текстом и шрифтами, в этом примере мы увидим дополнительные аспекты программирования Windows-приложений. Исходный текст программы (назовем ее `SCOL`) представлен в листинге 5.9.

Листинг 5.9. Программа SCOL, изменяющая цвет шрифта текста в окне приложения

```
;----------------------- SCOL.ASM -----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib
```

```
  ;--------    WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB  "ВЫВОД СТРОК ТЕКСТА РАЗНЫМИ ШРИФТАМИ И ЦВЕТОМ", 0
  CommandLine   DD  0
  hWnd          DD  0
  hInstance     DD  0
  szClassName   DB  "Demo_Class", 0

  myStr         label BYTE
  s1            DB  " STRING 1", 0
  s2            DB  "  STRING 2", 0
  s3            DB  "   STRING 3", 0
  ls            DD  0
   ; Атрибуты шрифта

  myf           DB  "Arial Cyr", 0
  mypitch       EQU DEFAULT_PITCH or FF_SWISS
  myq           EQU DEFAULT_QUALITY
  myclip        EQU CLIP_DEFAULT_PRECIS
  myout         EQU OUT_DEFAULT_PRECIS
  myansi        EQU ANSI_CHARSET

  vHeight       DD  0              ; переменная, хранящая высоту шрифта
  colorRef      DD  0F1AA7Ch       ; начальное значение цвета шрифта

.code

start:
 push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX
  call    GetCommandLine
  mov     CommandLine, EAX
   ; Устанавливаем начальное значение высоты шрифта
   mov     vHeight, 12
```

```
 push     SW_SHOWDEFAULT
 push     CommandLine
 push     NULL

 push     hInstance
 call     WinMain
 push     EAX
 call     ExitProcess
WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

 LOCAL wc   :WNDCLASSEX
 LOCAL msg  :MSG
  ;  Заполнение структуры WNDCLASSEX
  mov     wc.cbSize, sizeof WNDCLASSEX
 mov     wc.style, CS_HREDRAW or CS_VREDRAW
  mov     wc.lpfnWndProc, offset WndProc
 mov     wc.cbClsExtra, NULL
 mov     wc.cbWndExtra, NULL

 push     hInst
 pop      wc.hInstance

 mov      wc.hbrBackground, COLOR_BTNFACE+7
 mov      wc.lpszMenuName, NULL
 mov      wc.lpszClassName, offset szClassName
 push     IDI_APPLICATION
push     NULL
call     LoadIcon
 mov      wc.hIcon, EAX
 push     IDC_ARROW
 push     NULL
 call     LoadCursor
 mov      wc.hCursor, EAX
 mov      wc.hIconSm, 0

 lea      EAX, wc
```

```
  push     EAX
  call     RegisterClassEx

  push     NULL
  push     hInst
  push     NULL
  push     NULL

  push     CW_USEDEFAULT
  push     CW_USEDEFAULT
  push     CW_USEDEFAULT
  push     CW_USEDEFAULT
   push     WS_OVERLAPPEDWINDOW
  push     offset szDisplayName
  push     offset szClassName
  push     WS_EX_OVERLAPPEDWINDOW
  call     CreateWindowEx
  mov      hWnd,EAX

  push     SW_SHOWNORMAL
  push     hWnd
  call     ShowWindow

  push     hWnd
  call     UpdateWindow
 ; Цикл обработки сообщений

StartLoop:
  push     0
  push     0
  push     NULL
  lea      EAX, msg
  push     EAX
  call     GetMessage   cmp     EAX, 0
  je       ExitLoop

  lea      EAX, msg
  push     EAX
```

```
  call    TranslateMessage
  lea     EAX, msg
  push    EAX
  call    DispatchMessage

  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp
 ; Оконная процедура
 WndProc proc hWin    :DWORD,
              uMsg    :DWORD,
              wParam  :DWORD,
              lParam  :DWORD

  LOCAL hdc       :HDC
  LOCAL ps        :PAINTSTRUCT
  LOCAL rect      :RECT
  LOCAL myFont    :HFONT
  LOCAL tm        :TEXTMETRIC
  LOCAL tx, ty    :DWORD
  LOCAL cnt       :DWORD
  LOCAL x, y      :DWORD

  cmp     uMsg, WM_PAINT
  jne     next_1
  lea     ESI, ps
  push    ESI
  push    hWin
  call    BeginPaint
  mov     hdc, EAX
   invoke  SendMessage, hWin, WM_PRINT, hdc, PRF_CLIENT
  lea     ESI, ps
  push    ESI
  push    hWin
```

```
  call    EndPaint
  ret
next_1:
  cmp     uMsg, WM_PRINT
  jne     next_2
 invoke  GetDC, hWin
  mov     hdc, EAX
  mov     CL, 4   rol     colorRef, CL

  invoke  SetTextColor, hdc, colorRef
  invoke  GetClientRect, hWin, ADDR rect
  invoke  GetTextMetrics, hdc, ADDR tm
  mov     EAX, tm.tmHeight
  shl     EAX, 1
  mov     ty, EAX

  mov     EAX, rect.right
  sub     EAX, rect.left
  xor     EDX, EDX
  mov     ECX, 3
  div     ECX
  mov     x, EAX

  mov     EAX, rect.bottom
  sub     EAX, rect.top
  xor     EDX, EDX
  mov     ECX, 5
  div     ECX
  mov     y, EAX

  push    vHeight
  mov     cnt, 1
  lea     ESI, myStr
next_raw:
  invoke CreateFont, vHeight, 0, 0, 0, 600, 0, 0, 0, myansi, myout,\
                   myclip, myq, mypitch, ADDR myf
  mov     myFont, EAX
  invoke  SelectObject, hdc, myFont   push    ESI
```

```
  call    LenStr
  mov     ls, EAX

  push    ls
  push    ESI
  push    y
  push    x
  push    hdc
  call    TextOut
  add     vHeight, 3

  mov     EAX, ty
  add     y, EAX

  add     ESI, ls
  inc     ESI
  cmp     cnt, 3
  je      cont
  inc     cnt
 jmp     next_raw
cont:
  invoke  ReleaseDC, hWin, hdc
  pop     vHeight
  ret

next_2:
  cmp     uMsg, WM_RBUTTONDOWN
  jne     next_3
  invoke  GetDC, hWin
  mov     hdc, EAX
  invoke  SendMessage, hWin, WM_PRINT, hdc, PRF_CLIENT
  invoke  ReleaseDC, hWin, hdc
  ret

next_3:
  cmp     uMsg, WM_DESTROY
  jne     next_4
  push    NULL
```

```
  call    PostQuitMessage
  xor     EAX, EAX
  ret

next_4:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  WndProc endp

; Процедура LenStr для определения размера строки

LenStr proc
  push    EBP
  mov     EBP, ESP
  mov     EDX, [EBP+8]
  mov     EDI, EDX
  cld
  mov     AL, 0
next_check:
  scasb
  je      ex
  jmp     next_check
ex:
  sub     EDI, EDX
  mov     EAX, EDI
  dec     EAX
  pop     EBP
  ret     4
LenStr endp
end start
```

Исходный текст программы довольно сложен, и перед его анализом имеет смысл рассмотреть более подробно структуру программы в целом. Поскольку функция `WinMain` является стандартной, рассматривать ее мы не будем. Нас будет интересовать только работа оконной процедуры `WndProc`. Процедура содержит несколько обработчиков сообщений. Рассмотрим их более

подробно как в плане реализации программного кода, так и в плане взаимодействия друг с другом и с операционной системой.

- Обработчик сообщения `WM_PAINT` выполняет обновление (перерисовку) окна приложения при инициализации программы. При изменениях размера окна или при перекрытии окном другого приложения операционная система генерирует сообшение `WM_PAINT`. Чтобы обеспечить прорисовку окна, программный код обработчика посылает сообщение `WM_PRINT` нашему приложению.
- Обработчик сообщения `WM_RBUTTONDOWN` вызывается при нажатии на правую кнопку мыши. При этом приложение должно изменить вид строк текста в клиентской области окна. Чтобы обеспечить такую возможность, программный код обработчика посылает сообщение `WM_PRINT` нашему приложению.
- Обработчик сообщения `WM_PRINT`, так же как и `WM_PAINT`, выполняет перерисовку клиентской области окна приложения. Оконная процедура передает управление обработчику при получении соответствующего сообщения от операционной системы. Кроме этого, здесь же выполняется вся обработка текстовых строк, включая установки цвета и шрифтов.

Новым для нас является то, что оконная процедура может посылать сообщения сама себе. Этот прием очень часто используется программистами. Реализуется такой подход довольно просто — необходимо в нужном месте оконной процедуры послать сообщение. Заметим, что посылать сообщения можно не только самим себе, но и другому приложению! Для этих целей служит функция WIN API `SendMessage`, принимающая в качестве параметров дескриптор окна приложения, которому посылается сообщение, идентификатор (тип) сообщения, первый параметр сообщения (`wParam`) и второй параметр сообщения (`lParam`). Функция `SendMessage` имеет следующий синтаксис:

```
LRESULT SendMessage (HWND    hWnd,        // дескриптор окна, которому
                                          // посылается сообщение
                     UINT    Msg,         // сообщение
                     WPARAM  wParam,      // первый параметр
                     LPARAM  lParam);     // второй параметр
```

Следует иметь в виду то, что для каждого сообщения определены свои параметры. Для `WM_PRINT` вызов функции `SendMessage` выполняется также, как, например, в обработчике `WM_RBUTTONDOWN`:

```
cmp     uMsg, WM_RBUTTONDOWN
jne     next_3
  ...
```

```
invoke  SendMessage, hWin, WM_PRINT, hdc, PRF_CLIENT
  ...
```

Здесь в качестве первого параметра выступает дескриптор контекста hdc, а в качестве второго — константа PRF_CLIENT, означающая, что необходимо перерисовать клиентскую область окна.

После анализа структуры программного кода детально рассмотрим обработчик сообщения WM_PRINT, который, как мы знаем, выполняет всю основную работу с текстом и шрифтами. Нас будет интересовать работа функции SetTextColor в этом обработчике. Фрагмент исходного текста, где вызывается эта функция:

```
colorRef DD 0F1AA7Ch
  ...
mov     CL, 4
rol     colorRef, CL
invoke  SetTextColor, hdc, colorRef
```

Переменная colorRef в секции объявления данных определяет цветовую комбинацию. В двойном слове значения первых трех байт определяют относительную интенсивность цветов:

- байт 0 (7Ch) — красного;
- байт 1 (AAh) — зеленого;
- байт 2 (F1h) — синего.

Старший байт должен быть равен 0. Я задал произвольные значения интенсивности цветов, а для демонстрации изменения цвета текста использовал команду циклического сдвига ror ассемблера. Цвет текстовых строк будет меняться при нажатии правой кнопки мыши, изменении размеров окна приложения, восстановлении окна приложения после перекрытия.

Далее приводится вариант оконной процедуры этой программы, разработанный с использованием только ассемблерных команд. Команды, смысл которых не очевиден, снабжены комментариями. Исходный текст процедуры WndProc приведен в листинге 5.10.

Листинг 5.10. Оконная процедура, использующая только ассемблерные команды

```
WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
```

```
  LOCAL hdc      :HDC
  LOCAL ps       :PAINTSTRUCT
  LOCAL rect     :RECT
  LOCAL myFont   :HFONT

  LOCAL tm       :TEXTMETRIC
  LOCAL tx, ty   :DWORD
  LOCAL cnt      :DWORD
  LOCAL x, y     :DWORD

  cmp     uMsg, WM_PAINT
  jne     next_1
  lea     ESI, ps
  push    ESI
  push    hWin
  call    BeginPaint
  mov     hdc, EAX
   push    PRF_CLIENT
  push    hdc    push    WM_PRINT
  push    [EBP+8]                       ; дескриптор окна hWin
  call    SendMessage
   lea     ESI, ps
  push    ESI
  push    hWin
  call    EndPaint
  ret
next_1:
  cmp     DWORD PTR [EBP+12], WM_PRINT   ;второй параметр по смещению +12
  jne     next_2

  push    [EBP+8]   call    GetDC   mov     hdc, EAX
  mov     CL, 4
  rol     colorRef, CL

  push    colorRef
  push    hdc
  call    SetTextColor
```

```
lea     ESI,rect
push    ESI
push    [EBP+8]
call    GetClientRect
lea     ESI,tm
push    ESI
push    hdc
call    GetTextMetrics
mov     EAX, tm.tmHeight
                        mov     ECX, 3

div     ECX
mov     x, EAX
mov     EAX, rect.bottom
sub     EAX, rect.top
xor     EDX, EDX

mov     ECX, 5
div     ECX
mov     y, EAX
push    vHeight
mov     cnt, 1

lea     ESI, myStr
push    offset myf
push    mypitch
push    myq

push    myclip
push    myout
push    myansi
push    0
push    0

push    0
push    600
push    0
push    0
push    0
```

```
  push    vHeight
  call    CreateFont         push    myFont

  push    hdc
  call    SelectObject   push    ESI
  call    LenStr
  mov     ls, EAX
  push    ls
  push    ESI

  push    y
  push    x
  push    hdc
  call    TextOut
  add     vHeight, 3
   mov     EAX, ty
  add     y, EAX
  add     ESI, ls
  inc     ESI
  cmp     cnt, 3

  je      cont
  inc     cnt

  jmp     next_raw
cont:
  push    hdc
  push    [EBP+8]
  call    ReleaseDC                DWORD PTR [EBP+12]         push
[EBP+8]   call    GetDC          push    PRF_CLIENT
  push    hdc
  push    WM_PRINT
  push    [EBP+8]

  call    SendMessage   push    hdc
  push    [EBP+8]
  call    ReleaseDC
               push    NULL
```

```
  call    PostQuitMessage
  xor     EAX, EAX
  ret

next_4:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin

  call    DefWindowProc
  ret
WndProc endp
```

Окна приложения при двух последовательных нажатиях правой кнопки мыши изображены на рис. 5.7 и 5.8.

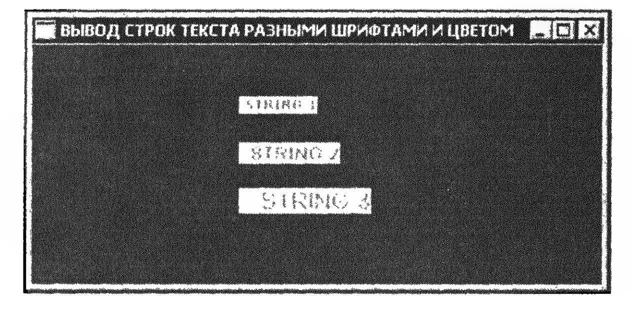


Рис. 5.7. Окно приложения, отображающего цвет шрифта при первом нажатии правой кнопки мыши

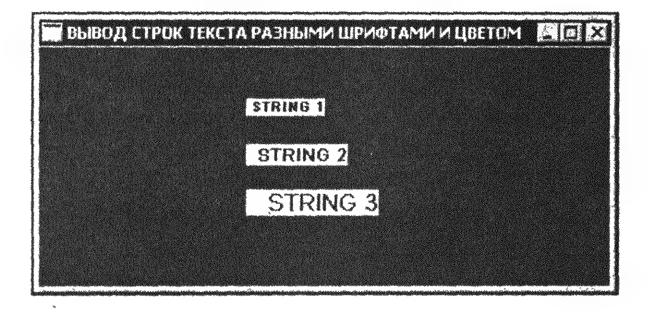


Рис. 5.8. Окно приложения, отображающего цвет шрифта при втором нажатии правой кнопки мыши

До сих пор мы уделяли внимание работе с основными атрибутами текстовых строк, такими как цвет и шрифт, а также рассмотрели вопросы позиционирования текста в окне приложения. Однако не все данные могут быть выведены на экран с помощью функции `TextOut`. К примеру, вывести целое число без преобразования его в текстовую строку невозможно. Как решить задачу конверсии числа в текст? Здесь на помощь приходит уже известная нам из *главе 2* функция WIN API `wsprintf`. Напомню ее синтаксис:

```
int wsprintf(LPTSTR    lpOut,          // выходной буфер
             LPCTSTR   lpFmt,          // строка форматирования
              ...                      // аргументы
             );
```

Функция `wsprintf` выполняет форматирование и запись последовательности символов и чисел в буфер. Все аргументы функции преобразуются и копируются в выходной буфер в соответствии со спецификацией, записанной в строке форматирования.

Функция добавляет завершающий ноль для записанных строк, однако возвращаемый результат (размер строки) не учитывает его. Должен заметить, что в Windows, к сожалению, нет функций для преобразования вещественных чисел в строку символов, подобных `wsprintf`.

Разработаем приложение, демонстрирующее преобразование целочисленного значения в строку и вывод ее в окно приложения. Пусть наша программа отображает количество щелчков правой кнопки мыши в окне приложения.

Исходный текст приложения (назовем его `CLICKS`) приведен в листинге 5.11.

Листинг 5.11 Программа, отображающая количество щелчков правой кнопки мыши

```
;-------------------- CLICKS.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
```

```
  includelib \masm32\lib\gdi32.lib

  ;--------    WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB "ПРИМЕР ИСПОЛЬЗОВАНИЯ ФУНКЦИИ wsprintf", 0
  CommandLine   DD 0
  hWnd          DD 0
  hInstance     DD 0
  szClassName   DB "Demo_Class", 0
  s1            DB "Вы нажали правую кнопку мыши ", 0
  s2            DB " раз!", 0

;------  wsprintf------
   lpFmt         DB "%s%d%s", 0
  buf           DB 128 dup (0)
  ibuf          DD 0
  cnt           DD 0
   tsize         label DWORD
  crx           DD 0
  cry           DD 0
.code

start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX

  invoke  GetCommandLine
  mov     CommandLine, EAX
  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, 0
 WinMain proc hInst     :DWORD,
              hPrevInst :DWORD,
              CmdLine   :DWORD,
              CmdShow   :DWORD
```

```
 ; Локальные переменные процедуры
  LOCAL wc    :WNDCLASSEX
 LOCAL msg   :MSG
  ; Заполнение структуры WNDCLASSEX требуемыми параметрами
  mov     wc.cbSize, sizeof WNDCLASSEX
 mov     wc.style, CS_HREDRAW or CS_VREDRAW

 mov     wc.lpfnWndProc, offset WndProc
 mov     wc.cbClsExtra, NULL
 mov     wc.cbWndExtra, NULL
 push ,  hInst
 pop     wc.hInstance
  mov     wc.hbrBackground, COLOR_BTNFACE+7
 mov     wc.lpszMenuName, NULL
 mov     wc.lpszClassName, offset szClassName

 invoke  LoadIcon, NULL, IDI_APPLICATION
 mov     wc.hIcon, EAX
  invoke  LoadCursor, NULL, IDC_ARROW
 mov     wc.hCursor, EAX
 mov     wc.hIconSm, 0

 invoke  RegisterClassEx, ADDR wc
 invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                         ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                         CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                         CW_USEDEFAULT, NULL, NULL, hInst, NULL
 mov     hWnd, EAX
 invoke  ShowWindow, hWnd, SW_SHOWNORMAL
 invoke ' UpdateWindow, hWnd

; Цикл обработки сообщений

StartLoop:
 invoke  GetMessage, ADDR msg, NULL, 0, 0
 cmp     EAX, 0
 je      ExitLoop
```

```
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc    :HDC
  LOCAL rect   :RECT
  LOCAL ps     :PAINTSTRUCT
  LOCAL tx, ty :DWORD
  LOCAL x, y   :DWORD
   cmp     uMsg, WM_PAINT
  jne     next_0
  lea     EDX, ps
  push    EDX
  push    hWin
  call    BeginPaint
  mov     hdc, EAX

  push    PRF_CLIENT
  push    hdc
  push    WM_PRINT
  push    hWin
  call    SendMessage   lea     EDX, ps
  push    EDX
  push    hWin
  call    EndPaint
  ret

next_0:
    PRINT
  lea     ESI, s2
```

```
  push    ESI
  push    cnt
  lea     ESI, s1
  push    ESI
  lea     ESI, lpFmt
  push    ESI
  lea     ESI, buf
  push    ESI
  call    wsprintf
  add     ESP, 20
  mov     ibuf, EAX
  invoke  GetTextExtentPoint32, hdc, ADDR buf, ibuf, ADDR tsize
  mov     EAX, rect.right
  sub     EAX, rect.left
  sub     EAX, crx
  shr     EAX, 1
  mov     x, EAX

  mov     EAX, rect.bottom
  sub     EAX, rect.top
  sub     EAX, cry
  shr     EAX, 1
  mov     y, EAX   invoke  TextOut, hdc, x, y, ADDR buf, ibuf
  invoke  ReleaseDC, hWnd, hdc
  ret

next_1:
  cmp     uMsg, WM_RBUTTONDOWN
  jne     next_2
  inc     cnt
  push    PRF_CLIENT
  push    hdc
  push    WM_PRINT
  push    hWin
  call    SendMessage
  ret
```

```
next_2:
  cmp     uMsg, WM_DESTROY
  jne     next_3
  invoke  PostQuitMessage, NULL
  xor     EAX, EAX

  ret
next_3:
  invoke  DefWindowProc, hWin, uMsg, wParam, lParam
  ret
 end start
```

Вначале рассмотрим, как формируется строка текста для отображения в окне приложения. Формирование строки выпоняется функцией `wsprintf`. Вот фрагмент программного кода такого преобразования:

```
  ...
s1      DB "Вы нажали правую кнопку мыши ", 0
s2      DB " раз!", 0

lpFmt   DB "%s%d%s", 0
buf     DB 128 dup (0)
ibuf    DD 0
cnt     DD 0   ...
lea     ESI, s2
push    ESI
push    cnt
lea     ESI, s1
push    ESI
lea     ESI, lpFmt
push    ESI

lea     ESI, buf
push    ESI
call    wsprintf
add     ESP, 20
mov     ibuf, EAX   ...
```

Функция wsprintf в качестве первого параметра принимает адрес буфера для хранения результата преобразования. Буфер buf должен иметь достаточный размер для размещения в нем результата преобразования. Вторым параметром функции является адрес строки форматирования lpFmt.

Строка форматирования представляет собой последовательность управляющих символов для определения способа преобразования и вывода аргументов, которые представляют собой третий и последующие параметры.

Очень важно запомнить, что в отличие от подавляющего большинства функций WIN API, использующих соглашение stdcall передачи параметров, wsprintf обрабатывает параметры в соответствии с директивой cdecl! Поскольку освобождать стек в этом случае должна вызывающая программа, то необходима последующая команда:

```
add    ESP, 20
```

Так как функция принимает пять параметров по четыре байта, то вызывающая программа должна освободить 20 байт.

В нашем случае строка форматирования имеет вид "%s%d%s", в качестве параметров передаются строка s1, целое число cnt и строка s2. В качестве результата ibuf функция возвращает фактическое количество символов, записанное в буфер без учета завершающего нуля, который всегда добавляется в конец строки.

После того, как буфер символов сформирован, остается только вывести его посередине окна приложения. Вычисление координат x и y начала вывода текста выполняется функцией GetTextExtentPoint32, а сам вывод осуществляется при помощи функции TextOut. Все операции преобразования и отображения строки текста выполняются в обработчике WM_PRINT оконной процедуры.

Полностью ассемблерный вариант обработчиков WM_PAINT, WM_PRINT и WM_RBUTTONDOWN оконной процедуры приведен в листинге 5.12.

Листинг 5.12. Обработчики событий WM_PAINT, WM_PRINT и WM_RBUTTONDOWN, написанные полностью на ассемблере

```
. . .
cmp     DWORD PTR [EBP+12], WM_PAINT
jne     next_0
lea     EDX, ps
push    EDX
push    [EBP+8]
call    BeginPaint
```

```
  mov     hdc, EAX
  push    PRF_CLIENT
  push    hdc
  push    WM_PRINT
  push    [EBP+8]
  call    SendMessage   lea     EDX, ps
  push    EDX
  push    [EBP+8]
  call    EndPaint
  ret

next_0:
  cmp     DWORD PTR [EBP+12], WM_PRINT
  jne     next_1
  push    [EBP+8]
  call    GetDC
  mov     hdc, EAX

  lea     ESI, rect
  push    ESI
  push    [EBP+8]
  call    GetClientRect
  push    offset s2
  push    cnt
  push    offset s1
  push    offset lpFmt

  push    offset buf
  call    wsprintf
  add     ESP, 20
  mov     ibuf, EAX
  push    offset tsize
  push    ibuf
  push    offset buf
  push    hdc
```

```
call     GetTextExtentPoint32

mov      EAX, rect.right
sub      EAX, rect.left
sub      EAX, crx
shr      EAX, 1
mov      x, EAX
mov      EAX, rect.bottom
sub      EAX, rect.top
sub      EAX, cry
shr      EAX, 1
mov      y, EAX   push    ibuf   push    offset buf
push     y
push     x
push     hdc
call     TextOut
 push     hdc
push     [EBP+8]
call     ReleaseDC    next_1:
cmp      DWORD PTR [EBP+12], WM_RBUTTONDOWN
jne      next_2   inc     cnt    push    PRF_CLIENT
push     hdc   push    WM_PRINT
push     [EBP+8]
call     SendMessage
ret
 ...
```

Вид окна работающего приложения изображен на рис. 5.9.



Рис. 5.9. Окно приложения, отображающего количество щелчков правой кнопки мыши на окне

5.4. Рисование геометрических фигур

Процесс рисования простейших геометрических фигур в Windows требует вызова некоторых функций графического интерфейса Windows. Например, чтобы нарисовать линию в окне приложения, необходимо вызвать по крайней мере две функции — `MoveToEx` и `LineTo`. Функция `MoveToEx` перемещает указанную точку на позицию с другими координатами. Функция имеет следующий синтаксис:

```
BOOL MoveToEx(HDC      hdc,          // дескриптор контекста
              Int      X,            // x — координата новой позиции
              int      Y,            // y — координата новой позиции
              LPPOINT lpPoint);      // предыдущие координаты точки
```

Непосредственное рисование линии выполняется с помощью функции `LineTo`, которая в качестве параметров принимает дескриптор контекста устройства и координаты конечной точки. Функция имеет синтаксис:

```
BOOL LineTo(HDC        hdc,          // дескриптор контекста
            int        nXEnd,        // x - координата конечной точки
            int        nYEnd         // y - координата конечной точки
            );
```

Рассмотрим пример, в котором требуется нарисовать диагональ прямоугольника клиентской области. Начальная точка известна — это (0, 0), а координаты конечной (правый нижний угол) вычисляются так:

$$x = rect.right - rect.left,$$

$$y = rect.bottom - rect.top.$$

Исходный текст приложения приведен в листинге 5.13.

Листинг 5.13. Программа, рисующая диагональ клиентской области окна приложения

```
;------------------- DRAWLINE.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
```

```
include \masm32\include\kernel32.inc
include \masm32\include\gdi32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib

;--------    WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
 szDisplayName DB "РИСОВАНИЕ ДИАГОНАЛИ ПРЯМОУГОЛЬНИКА", 0
 CommandLine   DD 0
 hWnd          DD 0
 hInstance     DD 0
 szClassName   DB "Demo_Class", 0
.code
                    mov     CommandLine, EAX
      push    CommandLine
 push    NULL
 push    hInstance
 call    WinMain

 push    EAX
 call    ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
  ; Локальные переменные процедуры
  LOCAL wc    :WNDCLASSEX
 LOCAL msg  :MSG
  ; Заполнение структуры WNDCLASSEX требуемыми параметрами
  mov     wc.cbSize, sizeof WNDCLASSEX
 mov     wc.style, CS_HREDRAW or CS_VREDRAW

 mov     wc.lpfnWndProc, offset WndProc
 mov     wc.cbClsExtra, NULL
```

```
mov     wc.cbWndExtra, NULL

push    hInst
pop     wc.hInstance

mov     wc.hbrBackground, COLOR_BTNFACE+6
mov     wc.lpszMenuName, NULL
mov     wc.lpszClassName, offset szClassName

push    IDI_APPLICATION
  NULL                     call    LoadCursor
mov     wc.hCursor, EAX
mov     wc.hIconSm, 0
lea     EAX, wc
push    EAX
call    RegisterClassEx

push    NULL
push    hInst
push    NULL
push    NULL

push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT

push    WS_OVERLAPPEDWINDOW
push    offset szDisplayName
push    offset szClassName
push    WS_EX_OVERLAPPEDWINDOW
call    CreateWindowEx
mov     hWnd, EAX

push    hWnd
call    UpdateWindow
```

```
; Цикл обработки сообщений
 StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop
   lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage
  jmp     StartLoop
ExitLoop:   mov     EAX, msg.wParam
  ret
WinMain endp
 WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  ; Локальные переменные

  LOCAL hdc  :HDC
  LOCAL ps   :PAINTSTRUCT
  LOCAL rect :RECT
  LOCAL x, y :DWORD

  cmp     uMsg, WM_PAINT
  jne     next_1
   lea     EDX, ps
  push    EDX
```

```
    push    hWnd
    call    BeginPaint
    mov     hdc, EAX
    lea     ESI, rect
    push    ESI
    push    hWin
    call    GetClientRect

    push    0
    push    0
    push    0
    push    hdc
    call    MoveToEx
    mov     EAX, rect.bottom
    sub     EAX, rect.top
    push    EAX

    mov     EAX, rect.right
    sub     EAX, rect.left
    push    EAX
    push    hdc
    call    LineTo
    lea     EDX, ps
    push    EDX
    push    hWnd
    call    EndPaint
    ret
next_1:
    cmp     uMsg, WM_DESTROY
    jne     next_2
    push    NULL

    call    PostQuitMessage
    xor     EAX, EAX
    ret
next_2:
    push    lParam
    push    wParam
```

```
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret
WndProc endp
end start
```

Мне кажется, программный код приложения достаточно понятен и в комментариях не нуждается. Общий вид окна приложения изображен на рис. 5.10.

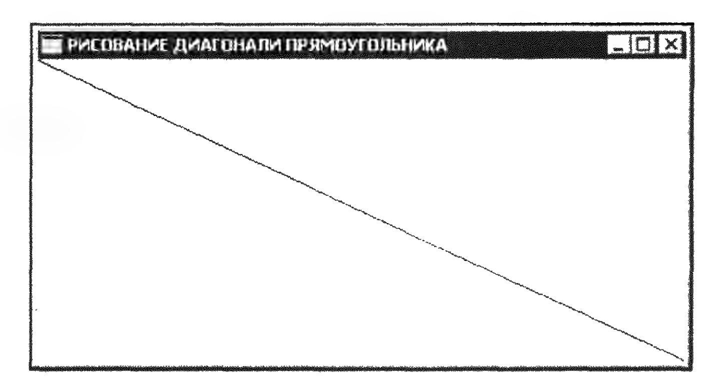


Рис. 5.10. Окно приложения, выполняющего рисование диагонали клиентской области

Попробуем нарисовать что-нибудь посложнее, например эллипс. Для рисования эллипса или окружности в заданной прямоугольной области предназначена функция `Ellipse`. Функция имеет следующий синтаксис:

```
BOOL Ellipse(HDC  hdc,             // дескриптор контекста устройства
             int  nLeftRect,       // координата x левого верхнего угла
                                   // ограничительного прямоугольника
             int  nTopRect,        // координата y левого вернего угла
             int  nRightRect,      // координата x правого нижнего угла
             int  nBottomRect      // координата y правого нижнего угла
             );
```

Рассмотрим простую программу, выполняющую рисование эллипса. Пусть центром эллипса является центр воображаемого прямоугольника, описываемого координатами левого верхнего (`x0, y0`) и правого нижнего (`x1, y1`) углов. Пусть координаты (`x0, y0`) отстоят от начала координат на $^1/_3$, а

координаты (x1, y1) — на $^2/_3$. Вывод изображения выполняется в обработчике WM_PAINT. Исходный текст программы (назовем ее DRAWEL) приведен в листинге 5.14.

Листинг 5.14. Программа, рисующая эллипс в клиентской области окна приложения

```
;---------------------- DRAWEL.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------    WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB "РИСОВАНИЕ ЭЛЛИПСА", 0
  CommandLine   DD 0
  hWnd          DD 0
  hInstance     DD 0
  szClassName   DB "Demo_Class", 0

.code

start:
            mov     hInstance, EAX
                                    call    WinMain
  push    EAX
  call    ExitProcess
```

```
WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

 LOCAL wc   :WNDCLASSEX
  LOCAL msg  :MSG
  ; Заполнение структуры WNDCLASSEX требуемыми параметрами

  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW

  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL

  push    hInst
  pop     wc.hInstance

  mov     wc.hbrBackground, COLOR_BTNFACE+9
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName

  push    IDI_APPLICATION
  push    NULL
  call    LoadIcon
  mov     wc.hIcon, EAX

  push    IDC_ARROW
  push    NULL
  call    LoadCursor
  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0
```

```
WinMain endp
 ;  Оконная процедура

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc  :HDC
  LOCAL ps   :PAINTSTRUCT
  LOCAL rect :RECT

  ; Переменные, где хранятся значения координат эллипса

  LOCAL x0, y0, x1, y1 :DWORD
   cmp     uMsg, WM_PAINT
  jne     next_1

  lea     EDX, ps
  push    EDX
  push    hWnd
  call    BeginPaint
  mov     hdc, EAX

  lea     ESI, rect
  push    ESI
  push    hWin
  call    GetClientRect
    Вычисление координаты x0

  mov     EAX, rect.right
  sub     EAX, rect.left
  push    EAX
  mov     EBX, 3
  xor     EDX, EDX

  div     EBX
  mov     ECX, rect.left
```

```
  lea     EAX, wc
  push    EAX
  call    RegisterClassEx

  invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL
                  call    ShowWindow

  push    hWnd
  call    UpdateWindow

; Цикл обработки сообщений

StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop
  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage
  jmp     StartLoop

ExitLoop:
  mov     EAX, msg.wParam
  ret
```

```
add     ECX, EAX
mov     x0, ECX

; Вычисление координаты x1

pop     EAX
mov     EBX, 3
xor     EDX, EDX
div     EBX
mov     ECX, rect.right

sub     ECX, EAX
mov     x1, ECX

;  Вычисление координаты y0

mov     EAX, rect.bottom
sub     EAX, rect.top
push    EAX
mov     EBX, 3
xor     EDX, EDX

div     EBX
mov     ECX, rect.top
add     ECX, EAX
mov     y0, ECX

; Вычисление координаты y1

pop     EAX
mov     EBX, 3
xor     EDX, EDX
div     EBX
mov     ECX, rect.bottom
sub     ECX, EAX
mov     y1, ECX

; Рисование эллипса
push    y1      x1      y0      x0          Ellipse   lea     EDX, ps
```

```
  push    EDX
  push    hWnd
  call    EndPaint
  ret
next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
                 next_2:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret
WndProc endp
end start
```

Окно работающего приложения изображено на рис. 5.11.

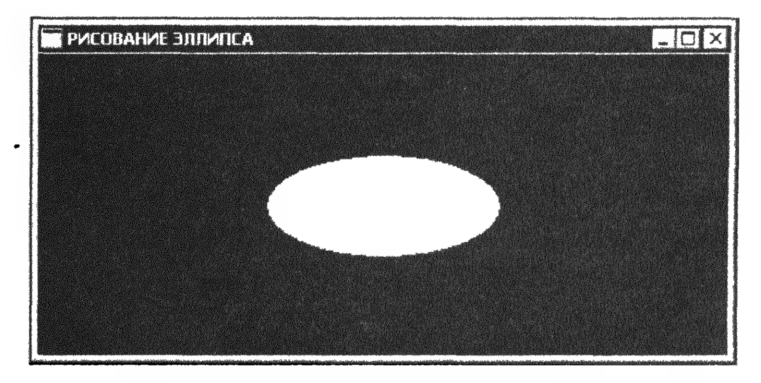


Рис. 5.11. Окно приложения, отображающего эллипс на экране

Исходный текст программы не очень сложен. Координаты эллипса вычисляются относительно левого вернего угла клиентской области окна. При изменении размеров окна расположение эллипса относительно начала координат сохраняется. Исходный текст в том виде, в каком он представлен, не очень хорошо структурирован. Сейчас мы попробуем воспользоваться теми возможностями, которые предоставляет нам макроассемблер MASM для улучшения качества программного кода и его читабельности. На этом примере мы рассмотрим различные варианты работы с процедурами и проанализируем каждый из них.

Первое, что можно сделать — выделить фрагменты кода, выполняющие однотипные вычисления, в отдельные процедуры. Такими фрагментами кода являются:

```
  ...
  Вычисление координаты x0

mov     EAX, rect.right
sub     EAX, rect.left
push    EAX
mov     EBX, 3
xor     EDX, EDX

div     EBX
mov     ECX, rect.left
add     ECX, EAX
mov     x0, ECX
  ...
```

а также:

```
  ...
; Вычисление координаты y0

mov     EAX, rect.bottom
sub     EAX, rect.top
push    EAX
mov     EBX, 3
xor     EDX, EDX

div     EBX
mov     ECX, rect.top
add     ECX, EAX
mov     y0, ECX
  ...
```

Объединим эти два фрагмента кода так, чтобы вычисления можно было выполнять одной процедурой. Исходный текст такой процедуры (назовем ее `CalcLeftTop`) представлен далее.

```
CalcLeftTop proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, [EBP+12]   sub     EAX, [EBP+8]
   mov     EBX, 3
  xor     EDX, EDX
  div     EBX    mov     ECX, [EBP+8]

  add     ECX, EAX
  mov     EAX, ECX
  pop     EBP
  ret     8
CalcLeftTop endp
```

Аналогично можно представить в виде процедуры и фрагменты, при помощи которых вычисляются координаты `x1` и `y1`. Исходный текст процедуры (назовем ее `CalcRightBottom`) представлен следующим фрагментом кода:

```
CalcRightBottom proc   push    EBP
  mov     EBP, ESP
  mov     EAX, [EBP+12]
  sub     EAX, [EBP+8]
  mov     EBX, 3

  xor     EDX, EDX
  div     EBX
  mov     ECX, [EBP+12]
  sub     ECX, EAX

  mov     EAX, ECX
  pop     EBP
  ret     8 CalcRightBottom endp
```

Исходный текст обработчика сообщения `WM_PAINT` изменится и будет выглядеть так, как приведено в листинге 5.15.

Листинг 5.15. Обработчик сообщения `WM_PAINT`, в котором используются процедуры

```
 ...
cmp     uMsg, WM_PAINT
jne     next_1

lea     EDX, ps
push    EDX
push    hWnd
call    BeginPaint
mov     hdc, EAX

lea     ESI, rect
push    ESI
push    hWin
call    GetClientRect

; Вычисление координаты x0

push    rect.right
push    rect.left
call    CalcLeftTop
mov     x0, EAX

; Вычисление координаты y0

push    rect.bottom
push    rect.top
call    CalcLeftTop
mov     y0, EAX

; Вычисление координаты x1

push    rect.right
push    rect.left
```

```
call    CalcRightBottom
mov     x1, EAX

; Вычисление координаты y1

push    rect.bottom
push    rect.top
call    CalcRightBottom
mov     y1, EAX

push    y1   x1   y0   x0      Ellipse
lea     EDX, ps
push    EDX
push    hWnd
call    EndPaint
ret
  ...
```

В этом фрагменте кода мы передаем параметры в процедуры в соответствии с соглашением `stdcall`.

Мы можем сделать следующий шаг и применить для вызова наших процедур `CalcLeftTop` и `CalcRightBottom` оператор `invoke`. Для этого необходимо внести соответствующие коррективы в текст программы. Исходный текст программы, в которой используется оператор `invoke`, приведен в листинге 5.16. Все изменения и коррективы выделены жирным шрифтом.

Листинг 5.16. Программа рисования эллипса, в которой используются разработанные процедуры и оператор `invoke`

```
;-------------------- DRAWEL.ASM -------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
```

```
LOCAL msg  :MSG

; Заполнение структуры WNDCLASSEX требуемыми параметрами

mov     wc.cbSize, sizeof WNDCLASSEX
mov     wc.style, CS_HREDRAW or CS_VREDRAW

mov     wc.lpfnWndProc, offset WndProc
mov     wc.cbClsExtra, NULL
mov     wc.cbWndExtra, NULL

push    hInst
pop     wc.hInstance

mov     wc.hbrBackground, COLOR_BTNFACE+1
mov     wc.lpszMenuName, NULL
mov     wc.lpszClassName, offset szClassName

push    IDI_APPLICATION
push    NULL
call    LoadIcon
mov     wc.hIcon, EAX

push    IDC_ARROW
push    NULL
call    LoadCursor
mov     wc.hCursor, EAX
mov     wc.hIconSm, 0
lea     EAX, wc
push    EAX
call    RegisterClassEx

invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                        ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                        CW_USEDEFAULT, NULL, NULL, hInst, NULL
mov     hWnd,EAX

push    SW_SHOWNORMAL
```

```
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------    WinMain          PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc          PROTO :DWORD,:DWORD,:DWORD,:DWORD
  CalcLeftTop      PROTO :DWORD,:DWORD
  CalcRightBottom  PROTO :DWORD,:DWORD

.data
  szDisplayName  DB "РИСОВАНИЕ ЭЛЛИПСА", 0
  CommandLine    DD 0
  hWnd           DD 0
  hInstance      DD 0
  szClassName    DD "Demo_Class", 0
.code

start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX   call  GetCommandLine
  mov   CommandLine, EAX

  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain

  push    EAX
  call    ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
   ; Локальные переменные процедуры
   LOCAL wc     :WNDCLASSEX
```

```
  push     hWnd
  call     ShowWindow

  push     hWnd    call     UpdateWindow

; Цикл обработки сообщений

StartLoop:
  push     0
  push     0
  push     NULL
  lea      EAX, msg
  push     EAX
  call     GetMessage

  cmp      EAX, 0
  je       ExitLoop
  lea      EAX, msg
  push     EAX
  call     TranslateMessage

  lea      EAX, msg
  push     EAX
  call     DispatchMessage

  jmp      StartLoop
ExitLoop:
  mov      EAX, msg.wParam
  ret
WinMain endp

WndProc proc hWin    :DWORD,
             uMsg    :DWORD,
             wParam  :DWORD,
             lParam  :DWORD

  LOCAL    hdc               :HDC
  LOCAL    ps                :PAINTSTRUCT
```

```
  LOCAL   rect           :RECT
  LOCAL   x0, y0, x1, y1 :DWORD

  cmp     uMsg, WM_PAINT
  jne     next_1

  lea     EDX, ps
  push    EDX
  push    hWnd
  call    BeginPaint
  mov     hdc, EAX
   lea     ESI, rect
  push    ESI
  push    hWin
  call    GetClientRect
   invoke  CalcLeftTop, rect.left, rect.right
  mov     x0, EAX
  invoke  CalcLeftTop, rect.top, rect.bottom
  mov     y0, EAX
  invoke  CalcRightBottom, rect.left, rect.right
  mov     x1, EAX
  invoke  CalcRightBottom, rect.top, rect.bottom
  mov     y1, EAX

  push    y1
  push    x1
  push    y0
  push    x0
  push    hdc
  call    Ellipse
  lea     EDX, ps
  push    EDX
  push    hWnd
  call    EndPaint
  ret
next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
```

```
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret
next_2:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret

WndProc endp

CalcLeftTop proc top1: DWORD, top2: DWORD
  mov     EAX, top2
  sub     EAX, top1
  mov     EBX, 3
  xor     EDX, EDX
  div     EBX
  mov     ECX, top1
  add     ECX, EAX
  mov     EAX, ECX
  ret     8
CalcLeftTop endp

CalcRightBottom proc bottom1:DWORD, bottom2:DWORD
  mov     EAX, bottom2
  sub     EAX, bottom1
  mov     EBX, 3
  xor     EDX, EDX
  div     EBX
  mov     ECX, bottom2
  sub     ECX, EAX
  mov     EAX, ECX
  ret     8
CalcRightBottom endp

end start
```

Операционная система Windows обладает весьма мощным набором функций, выполняющих рисование геометрических фигур. На основе приведенных примеров при определенной фантазии можно достаточно легко создавать весьма сложные фигуры и композиции.

5.5. Обработка сообщений мыши

В предыдущих примерах мы использовали два обработчика сообщений, поступающих от мыши, — WM_LBUTTONDOWN (нажата левая кнопка мыши) и WM_RBUTTONDOWN (нажата правая кнопка мыши). От мыши может поступать намного больше сообщений. Для программистов представляют интерес следующие события:

- WM_LBUTTONDOWN — нажата левая кнопка мыши;
- WM_LBUTTONUP — отпущена левая кнопка мыши;
- WM_RBUTTONDOWN — нажата правая кнопка мыши;
- WM_RBUTTONUP — отпущена правая кнопка мыши;
- WM_LBUTTONDBLCLK — двойной щелчок левой кнопкой мыши;
- WM_MOUSEMOVE — мышь перемещается по рабочей области окна.

Для всех этих сообщений в параметре lParam содержится положение курсора мыши. Младшее слово — это координата x, а старшее слово — координата y относительно верхнего левого угла рабочей области окна.

Рассмотрим пример программы, в которой текущие координаты курсора мыши отображаются в окне приложения. Исходный текст приложения (назовем его MPOINT) приведен далее в листинге 5.17.

Листинг 5.17. Программа, выводящая в окно приложения текущие координаты курсора мыши

```
;--------------------- MPOINT.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
```

```
  includelib \masm32\lib\gdi32.lib

  ;--------
  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB "ОТОБРАЖЕНИЕ КООРДИНАТ КУРСОРА МЫШИ", 0
  CommandLine   DD 0
  hWnd          DD 0
  hInstance     DD 0
  szClassName   DB "Demo_Class", 0

  ; Здесь сохраняем текущие координаты мыши
   point_x       DD 0
  point_y       DD 0
   ; Параметры для функции wsprintf
   sx            DB " X= ", 0
  sy            DB " Y= ", 0
  buf           DB 32 dup (0)
  ibuf          DD 0
  lpFmt         DB "%s%u%s%u", 0
 .code

start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX

  call    GetCommandLine
  mov     CommandLine, EAX

  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain
  push    EAX
```

```
  call    ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc  :WNDCLASSEX
  LOCAL msg :MSG
  ; Заполнение структуры WNDCLASSEX требуемыми параметрами
  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW

  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL

  push    hInst
  pop     wc.hInstance
  mov     wc.hbrBackground, COLOR_BTNFACE+1
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName

  invoke  LoadIcon, NULL, IDI_APPLICATION

  mov     wc.hIcon, EAX
  invoke  LoadCursor, NULL, IDC_ARROW

  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0

  invoke  RegisterClassEx, ADDR wc

  invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
```

```
                              CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                              CW_USEDEFAULT, NULL, NULL, hInst, NULL
  mov     hWnd, EAX
  invoke  ShowWindow, hWnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hWnd

; Цикл обработки сообщений

StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop
  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage

  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

; Оконная процедура

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
```

```
            wParam :DWORD,
            lParam :DWORD

  LOCAL hdc  :HDC
  LOCAL ps   :PAINTSTRUCT
  LOCAL rect :RECT

  cmp     uMsg, WM_PAINT
  jne     next_1
  invoke  BeginPaint, hWnd, ADDR ps
  mov     hdc, EAX
  invoke  TextOut, hdc, point_x, point_y, ADDR buf, ibuf
  lea     EDX, ps
  push    EDX
  push    hWnd
  call    EndPaint
  ret

next_1:
  cmp     uMsg, WM_MOUSEMOVE
  jne     next_2

  invoke  GetDC, hWin
  mov     hdc, EAX
  invoke  GetClientRect, hWin, ADDR rect

  mov     EAX, lParam
  mov     point_x, EAX
  and     point_x, 0FFFFh
  shr     EAX, 16
  mov     point_y, EAX
   push    point_y
  push    offset sy
  push    point_x
  push    offset sx
  push    offset lpFmt

  push    offset buf
  call    wsprintf
```

```
  add     ESP, 24
  mov     ibuf, EAX

  invoke  InvalidateRect, hWin, ADDR rect, TRUE
  push    hdc
  push    hWin
  call    ReleaseDC
  ret

next_2:
  cmp     uMsg, WM_DESTROY
  jne     next_3
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret

next_3:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret

WndProc endp

end start
```

Проанализируем обработчики `WM_MOUSEMOVE` и `WM_POINT` оконной процедуры приложения, т. к. именно здесь и выполняются все манипуляции по считыванию и отображению координат курсора мыши. Как мы знаем, все сообщения мыши содержат в `lParam` координаты курсора.

Координату `x` сохраним в переменной `point_x`, а координату `y` — в переменной `point_y`. Это делается в обработчике `WM_MOUSEMOVE` при помощи следующего фрагмента кода:

```
  ...
mov     EAX, lParam
mov     point_x, EAX
```

```
and     point_x, 0FFFFh

shr     EAX, 16
mov     point_y, EAX
```

После того, как координаты вычислены и сохранены, выполняем формирование текстовой строки результата при помощи функции WIN API `wsprintf`. Строку текста сохраним в символьном буфере `buf`, а ее размер — в переменной `ibuf`. Эти переменные нужны для вызова функции `TextOut`. Поскольку функция `wsprintf` вызывается в соответствии с директивой `cdecl`, важно не забыть освободить стек после вызова функции. Это все выполняется с помощью следующего кода:

```
  ...
push    point_y
push    offset sy
push    point_x
push    offset sx
push    offset lpFmt

push    offset buf
call    wsprintf
add     ESP, 24
mov     ibuf, EAX
  ...
```

Наконец, для перерисовки окна приложения выполняем вызов функции `InvalidateRect`.

Задача обработчика сообщения `WM_PAINT` — вывести текстовую строку с координатами курсора мыши в окно приложения. Функция `TextOut`, выполняющая эту операцию, в качестве координат принимает значения `point_x` и `point_y`, вычисленные ранее.

Следующий пример показывает, каким образом можно рисовать при помощи мыши. Это не самый лучший способ рисования, но на нем можно отследить некоторые особенности работы устройства мыши. Приложение обрабатывает сообщения `WM_MOUSEMOVE`, `WM_RBUTTONDOWN`, `WM_LBUTTONDOWN` и `WM_LBUTTONUP`. Исходный текст приложения (назовем его `MDRAW`) приводится в листинге 5.18.

Листинг 5.18. Программа, выполняющая рисование в окне приложения с помощью мыши

```
;--------------------- MDRAW.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------
  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB "РИСОВАНИЕ ПРИ ПОМОЩИ МЫШИ", 0
  CommandLine   DD 0
  hWnd          DD 0
  hInstance     DD 0
  szClassName   DB "Demo_Class", 0

  ; Здесь сохраняем начальные координаты курсора мыши

  prev_x        DD 100
  prev_y        DD 100

  ; Здесь сохраняем текущие координаты курсора мыши

  cur_x         DD 0
  cur_y         DD 0

  ; Флаг переключения режимов
```

```
    toggle        DD 0
.code
start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX

  call    GetCommandLine
  mov     CommandLine, EAX

  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain

  push    EAX
  call    ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc    :WNDCLASSEX
  LOCAL msg   :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами

  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW

  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
```

```
  mov     wc.cbWndExtra, NULL
push    hInst
  pop     wc.hInstance

  mov     wc.hbrBackground, COLOR_BTNFACE+5
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName
   invoke  LoadIcon, NULL, IDI_APPLICATION

  mov     wc.hIcon, EAX
  invoke  LoadCursor, NULL, IDC_ARROW

  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0
  invoke  RegisterClassEx, ADDR wc

  invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW,ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL
  mov     hWnd, EAX
  invoke  ShowWindow, hWnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hWnd

; Цикл обработки сообщений

StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage
  cmp     EAX, 0
  je      ExitLoop
```

```
  lea     EAX, msg
  push    EAX
  call    TranslateMessage
  lea     EAX, msg
  push    EAX
  call    DispatchMessage
  jmp     StartLoop ExitLoop:
  mov     EAX, msg.wParam
  ret

WinMain endp

;  Оконная процедура

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc   :HDC
  LOCAL ps    :PAINTSTRUCT
  LOCAL rect  :RECT
  LOCAL point :POINT

 cmp     uMsg, WM_PAINT
  jne     next_1
  invoke  BeginPaint, hWnd, ADDR ps
  mov     hdc, EAX
  lea     EDX, ps
  push    EDX
  push    hWnd
  call    EndPaint
  ret
next_1:
  cmp     uMsg, WM_MOUSEMOVE
  jne     next_2
```

```
  cmp     toggle, 0    ;если отпущена левая кнопка мыши, рисовать нельзя
  je      ex_wmmov
  invoke  GetDC, hWin
  mov     hdc, EAX
  invoke  GetClientRect, hWin, ADDR rect

  ; Сохранение текущих координат мыши

  mov     EAX, lParam
  mov     cur_x, EAX
  and     cur_x, 0FFFFh
  shr     EAX, 16
  mov     cur_y, EAX   invoke  MoveToEx, hdc, prev_x, prev_y, 0
  invoke  LineTo, hdc, cur_x, cur_y
  invoke  ReleaseDC, hWin, hdc
ex_wmmov:
  ret
next_2:
  cmp     uMsg, WM_RBUTTONDOWN
  jne     next_3

  ; Очистка клиентской области приложения

  invoke  GetDC, hWin
  mov     hdc, EAX
  invoke  GetClientRect, hWin, ADDR rect
  invoke  InvalidateRect, hWin, ADDR rect, TRUE
  invoke  ReleaseDC, hWin, hdc

  ; Установка начальных координат рисования

  mov     prev_x, 100
  mov     prev_y, 100
  ret
next_3:
  cmp     uMsg, WM_LBUTTONDOWN
  jne     next_4
```

```
  ; Левая кнопка нажата, можно рисовать

  mov     toggle, 1

  ; Сместим начальную точку рисования относительно предыдущей

  add     prev_x, 100
  add     prev_y, 10
  ret
next_4:
  cmp     uMsg, WM_LBUTTONUP
  jne     next_5

  ; Левая кнопка отпущена, рисовать нельзя

  mov     toggle, 0
  ret
next_5:
  cmp     uMsg, WM_DESTROY
  jne     next_6
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret

next_6:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret

WndProc endp

end start
```

Программа рисует набор линий при нажатии и удержании левой кнопки мыши (сообщение WM_LBUTTONDOWN). Если кнопка отпущена (сообщение WM_LBUTTONUP), рисование прекращается. Отображение линий выполняется при перемещении мыши по клиентской области окна (сообщение WM_MOUSEMOVE).

Очистка клиентской области окна выполняется при нажатии правой кнопки мыши (сообщение WM_RBUTTONDOWN). При обработке этого сообщения восстанавливается и начальная точка рисования.

Если левая кнопка мыши была отпущена, то при повторном нажатии координаты начальной точки рисования меняются. Сам процесс рисования линий выполняется следующим фрагментом кода:

```
  ...
mov     EAX, lParam
mov     cur_x, EAX
and     cur_x, 0FFFFh
shr     EAX, 16
mov     cur_y, EAX
 invoke  MoveToEx, hdc, prev_x, prev_y, 0
invoke  LineTo, hdc, cur_x, cur_y
  ...
```

Первые пять команд вычисляют координаты курсора мыши, а следующие две (они нам уже знакомы) — рисуют линию. Окно работающего приложения изображено на рис. 5.12.



Рис. 5.12. Окно приложения, выполняющего рисование линий с помощью мыши

5.6. Ввод данных с клавиатуры

Несмотря на то, что большинство приложений активно использует мышь, без ввода информации с клавиатуры не обходится ни одна программа. Далее мы рассмотрим несколько примеров работы с клавиатурой, но вначале коротко о том, как Windows обрабатывает клавиатурный ввод.

Программа узнает о нажатиях клавиш посредством сообщений, которые посылаются оконной процедуре. Операционная система передает сообщения клавиатуры, по одному за раз, в очередь сообщений программы, содержащей окно с *фокусом ввода* (input focus). После этого программа отправляет сообщения соответствующей оконной процедуре.

Чтобы отобразить события клавиатуры, Windows посылает восемь различных сообщений программе. Приложение может игнорировать многие из них. Обычно такие сообщения содержат намного больше информации, чем нужно приложению. Рассмотрим наиболее важные из них.

Первое, что всегда интересует программиста — это как получить код нажатой клавиши.

Для нажатых символьных клавиш Windows генерирует сообщение `WM_CHAR`. Разработаем простое приложение, отображающее в клиентской области окна введенный с клавиатуры символ. Исходный текст приложения (назовем его `SHOWCH`) приведен в листинге 5.19.

Листинг 5.19. Программа, выводящая в окно приложения введенный с клавиатуры символ

```
;--------------------- SHOWCH.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------
```

```
  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB "ОТОБРАЖЕНИЕ ВВЕДЕННОГО С КЛАВИАТУРЫ СИМВОЛА", 0
  CommandLine   DD 0
  hWnd          DD 0
  hInstance     DD 0
  szClassName   DB "Demo_Class", 0
  sTitle        DB "Введенный символ", 0
  sMsg          DB "Вы ввели символ "
  c1            DB ?, 0

.code

start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX

  call    GetCommandLine
  mov     CommandLine, EAX
  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain
  push    EAX
  call    ExitProcess
 WinMain proc hInst     :DWORD,
              hPrevInst :DWORD,
              CmdLine   :DWORD,
              CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc   :WNDCLASSEX
  LOCAL msg  :MSG
```

```
; Заполнение структуры WNDCLASSEX требуемыми параметрами

mov     wc.cbSize, sizeof WNDCLASSEX
mov     wc.style, CS_HREDRAW or CS_VREDRAW
mov     wc.lpfnWndProc, offset WndProc
mov     wc.cbClsExtra, NULL
mov     wc.cbWndExtra, NULL

push    hInst
pop     wc.hInstance

mov     wc.hbrBackground, COLOR_BTNFACE+9
mov     wc.lpszMenuName, NULL
mov     wc.lpszClassName, offset szClassName
invoke  LoadIcon, NULL, IDI_APPLICATION
mov     wc.hIcon, EAX
invoke  LoadCursor, NULL, IDC_ARROW

mov     wc.hCursor, EAX
mov     wc.hIconSm, 0
invoke  RegisterClassEx, ADDR wc

invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                        ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                        CW_USEDEFAULT, NULL, NULL, hInst, NULL
mov     hWnd, EAX
invoke  ShowWindow,hWnd,SW_SHOWNORMAL
invoke  UpdateWindow,hWnd

; Цикл обработки сообщений

StartLoop:
 push   0
 push   0
 push   NULL
 lea    EAX, msg
```

```
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop

  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage

  jmp    StartLoop
ExitLoop:
  mov    EAX, msg.wParam
  ret

WinMain endp

; Оконная процедура

WndProc proc hWin    :DWORD,
             uMsg    :DWORD,
             wParam  :DWORD,
             lParam  :DWORD

  LOCAL hdc   :HDC
  LOCAL ps    :PAINTSTRUCT

  cmp     uMsg, WM_PAINT
  jne     next_1
  invoke  BeginPaint, hWnd, ADDR ps
  mov     hdc, EAX
  lea     EDX, ps
  push   EDX
  push   hWnd
```

```
  call   EndPaint
  ret
next_1:
  cmp     uMsg, WM_CHAR
  jne     next_2

  mov     EAX, wParam
  mov     cl, AL
  push    MB_OK
  push    offset sTitle
  push    offset sMsg
  push    0
  call    MessageBox
  ret
next_2:
  cmp     uMsg, WM_DESTROY
  jne     next_3
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret
next_3:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret

WndProc endp

end start
```

Проанализируем обработчик сообщения `WM_CHAR`. Если необходимо получить символ, соответствующий нажатой клавише, то наличие такого обработчика в оконной процедуре обязательно. При отправлении приложению такого сообщения операционная система Windows помещает в переменную `wParam` код нажатой клавиши, который может быть использован в дальнейшем. В данном случае программа помещает код символа в регистр `AL` и

затем сохраняет его в переменной cl, после чего выводит сообщение. Фрагмент кода, выполняющий эти операции, несложен и выглядит так:

```
  ...
mov     EAX, wParam
mov     cl, AL
push    MB_OK
push    offset sTitle
push    offset sMsg
push    0
call    MessageBox  ...
```

На рис. 5.13 изображено окно работающего приложения.

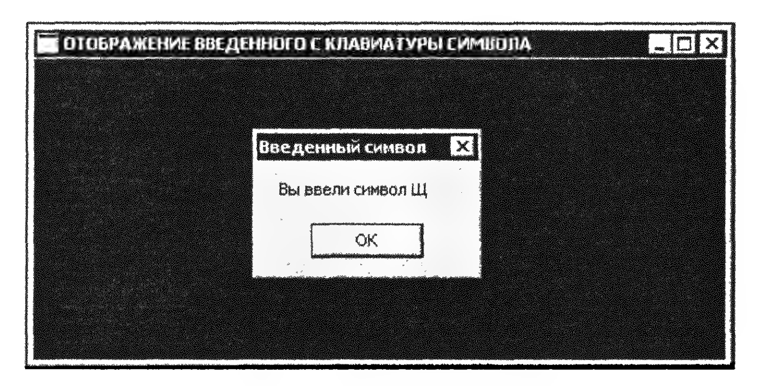


Рис. 5.13. Окно приложения, отображающего введенный с клавиатуры символ

Следующий пример более сложный. Проанализируем регистр введенных с клавиатуры символов, и строчные символы преобразуем в прописные. Вводимые символы отобразим в окне приложения. Для анализа и преобразования символов будем использовать две функции WIN API, с которыми мы еще не встречались — CharUpper и IsCharUpper. Функция CharUpper имеет следующий синтаксис:

```
LPTSTR CharUpper(LPTSTR lpsz          // указатель на символ или
                                      // строку с завершающим нулем
                );
```

Функция IsCharUpper определяет, принадлежит ли символ верхнему регистру. Анализ основывается на семантике выбранного пользователем языка

во время инсталляции системы или через панель управления Windows. Функция `IsCharUpper` имеет синтаксис:

```
BOOL IsCharUpper(TCHAR ch              // проверяемый символ
                );
```

Если символ оказывается прописной буквой, функция возвращает `TRUE`, в противном случае — `FALSE`.

Исходный текст программы (назовем ее SHOWC) приведен далее в листинге 5.20.

Листинг 5.20. Программа, преобразующая строчные символы в прописные

```
;---------------------- SHOWC.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------
  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB  "ПРЕОБРАЗОВАНИЕ СТРОЧНЫХ СИМВОЛОВ В ПРОПИСНЫЕ", 0
  CommandLine   DD  0
  hWnd          DD  0
  hInstance     DD  0
  szClassName   DB  "Demo_Class", 0
  s1            DB  "Введенный символ: ", 0
  ls            EQU $-s1-1
```

```
  dl            DB  "Преобразованный символ: ", 0
  ld            EQU $-dl-1 .code

start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX
  call    GetCommandLine
  mov     CommandLine, EAX   push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain

  push    EAX
  call    ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc    :WNDCLASSEX
  LOCAL msg   :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами

  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW

  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL

  push    hInst
  pop     wc.hInstance
```

```
  mov     wc.hbrBackground, COLOR_BTNFACE+5
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName
  invoke  LoadIcon, NULL, IDI_APPLICATION

  mov     wc.hIcon, EAX
  invoke  LoadCursor, NULL, IDC_ARROW

  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0
  invoke  RegisterClassEx, ADDR wc
   invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                           ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                           CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                           CW_USEDEFAULT, NULL, NULL, hInst, NULL
  mov     hWnd, EAX
  invoke  ShowWindow, hWnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hWnd

; Цикл обработки сообщений

StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage

  cmp     EAX, 0
  je      ExitLoop

  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
```

```
  push     EAX
  call     DispatchMessage

  jmp    StartLoop
ExitLoop:
  mov    EAX, msg.wParam
  ret
WinMain endp

;  Оконная процедура

WndProc proc hWin    :DWORD,
             uMsg    :DWORD,
             wParam :DWORD,
             lParam :DWORD

  LOCAL hdc   :HDC
  LOCAL ps    :PAINTSTRUCT
  LOCAL rect :RECT
  LOCAL x, y :DWORD

  cmp     uMsg, WM_PAINT
  jne     next_1

  invoke  BeginPaint, hWnd, ADDR ps
  mov     hdc, EAX

  invoke  GetClientRect, hWin, ADDR rect
  mov     EAX, rect.right
  sub     EAX, rect.left
  shr     EAX, 2
  mov     x,    EAX

  mov     EAX, rect.bottom
  sub     EAX, rect.top
  shr     EAX, 1
  mov     y,    EAX
   invoke  TextOut, hdc, x, y, ADDR s1,ls
```

```
  add     y, 20
  invoke  TextOut, hdc, x, y, ADDR dl,ld

  lea     EDX, ps
  push    EDX
  push    hWnd
  call    EndPaint
  ret
next_1:
  cmp     uMsg, WM_CHAR
  jne     next_2

  invoke  GetDC, hWin
  mov     hdc, EAX
  invoke  GetClientRect, hWin, ADDR rect   mov     EAX, wParam
  lea     EDI, s1
  add     EDI, ls
  sub     EDI, 2
  mov     [EDI], AL

  lea     EDI, dl
  add     EDI, ld
  sub     EDI, 2
  mov     [EDI], AL

  push    DWORD PTR [EDI]
  call    IsCharUpper   cmp     EAX, 1
  je      next
  push    EDI
  call    CharUpper next:
  invoke  InvalidateRect, hWin, ADDR rect, TRUE
  invoke  ReleaseDC, hWin, hdc
  ret
next_2:
  cmp     uMsg, WM_DESTROY
  jne     next_3
  push    NULL
```

```
  call    PostQuitMessage
  xor     EAX, EAX
  ret
next_3:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret

WndProc endp

end start
```

Анализ примера начнем с обработчика WM_CHAR. Как и в предыдущем примере, оконная процедура отслеживает ввод символов с клавиатуры через обработчик этого сообщения. Полученный символ извлекается из параметра сообщения wParam в регистр EAX. Наша программа должна отображать полученный и преобразованный символы через функцию TextOut в двух отдельных строках, поэтому сразу поместим полученный символ в буфер s1, а преобразованный — в d1. Для размещения символа в s1 необходимо выполнить следующие команды:

```
  ...
mov     EAX, wParam
lea     EDI, s1
add     EDI, ls
sub     EDI, 2
mov     [EDI], AL
  ...
```

Преобразованный символ размещается в строке d1 при помощи аналогичных команд, но вдобавок подвергается анализу с помощью команд CharUpper и IsCharUpper. Фрагмент кода будет следующим:

```
  ...
lea     EDI, d1
add     EDI, ld
```

```
sub     EDI, 2
mov     [EDI], AL

push    DWORD PTR [EDI]
call    IsCharUpper
cmp     EAX, 1
je      next

push    EDI
call    CharUpper
  ...
```

Функция IsCharUpper возвращает значение типа bool (TRUE или FALSE) в зависимости от результата анализа символа. Параметром вызова этой функции является значение символа. Функция IsCharUpper возвращает результат в регистре EAX. Если результат равен 1 (TRUE), то символ в преобразовании не нуждается, если равен 0 (FALSE), то вызывается функция CharUpper. Единственным параметром этой функции является адрес символа, который и помещается в стек перед вызовом функции. После того, как оба символа помещены в буферы s1 и d1, вызывается функция InvalidateRect, заставляющая приложение перерисовать окно. Это все, что касается обработчика WM_CHAR.

Вывод текстовых буферов выполняется в обработчике сообщения WM_PAINT при помощи функции TextOut обычным способом и в дополнительных пояснениях не нуждается.

Окно приложения изображено на рис. 5.14.

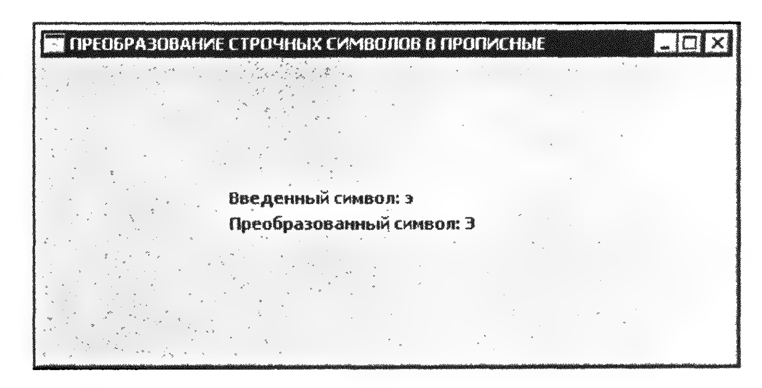


Рис. 5.14. Окно приложения, отображающего исходный и преобразованный символы

5.7. Элементы управления Windows и их применение в программах на ассемблере

Как бы мы ни использовали богатый арсенал функций WIN API и как бы ни комбинировали обработчики сообщений, добиться высокой функциональности приложения без элементов управления и ресурсов очень трудно.

Вначале рассмотрим возможности элемента управления меню. Это, вероятно, наиболее важная часть пользовательского интерфейса, который предлагают программы для Windows, а использование меню в программе — задача не сложная.

Прежде всего необходимо определить структуру меню в файле описания ресурсов и присвоить каждому пункту меню уникальный идентификатор. Имя меню должно быть определено в структуре класса окна. Когда в работающем приложении выбирается пункт меню, Windows посылает программе сообщение WM_COMMAND, содержащее этот идентификатор.

При запуске программы строка меню выводится на экране непосредственно под строкой заголовка. Эту строку иногда называют главным меню (Main Menu) или меню верхнего уровня (top-level menu). При выборе элемента главного меню раскрывается другое меню. Это меню называется выпадающим (popup menu). Иногда можно услышать другое название — подменю (submenu). Программист может определить несколько уровней вложенности для выпадающих меню. Пункты (опции) выпадающих меню могут быть помечены (checked), при этом слева от текста элемента меню ставится метка, определяющая, какие опции программы выбраны.

Пункты главного меню пометить нельзя. Кроме этого, пункты в главном и выпадающих меню могут быть "разрешены" (enabled), "запрещены" (disabled) или "недоступны" (grayed).

Создать меню можно несколькими способами. Наибольшее распространение получил способ определения меню в файле описания ресурсов в форме шаблона меню, например:

```
MyMenu MENU
{
  [список элементов меню]
}
```

где MyMenu — это имя элемента управления меню. Это имя должно присваиваться соответствующему полю в структуре класса окна. Более подробно структуру меню мы увидим немного позже при рассмотрении примеров.

Файл описания ресурсов должен быть предварительно откомпилирован редактором (компилятором) ресурсов. Для этого используется утилита rc.exe. Предварительно подготовленный файл (имеющий расширение RC) с описанием ресурсов компилируется в двоичный файл с расширением RES. Полученный файл при помощи компоновщика включается в исполняемый модуль приложения. Например, если приложение `myapp` использует ресурс, описанный в файле mymenu.rc, то для компиляции и сборки программы следует выполнить последовательность операторов

```
ml /c /coff /Cp myapp.asm
rc mymenu.rc
link /SUBSYSTEM:WINDOWS /LIBPATH:c:\masm32\lib myapp.obj mymenu.res
```

Каждый пункт меню должен иметь уникальный в пределах данного меню идентификатор (целое число). Windows посылает оконной процедуре идентификатор выбранного пункта меню в качестве параметра. Обычно вместо численных значений используют идентификаторы, определенные в разделе констант программного кода. Эти идентификаторы начинаются с символов `IDM`. Обозначения придуманы разработчиками Microsoft, хотя можно использовать и свои. Идентификаторы элементов управления начинаются с символов `IDC`. Последние символы (`M` и `C`) в префиксах `IDM` и `IDC` обозначают `Menu` и `Control` соответственно.

Объявление имени меню в классе окна является наиболее распространенным способом ссылки на ресурс меню, но часто применяют и другой вариант. Ресурс меню можно загрузить в память с помощью функции WIN API `LoadMenu`. Если в описании ресурса используется имя, то функция `LoadMenu` возвращает дескриптор меню:

```
   ...
.data
  MenuName DB "MyMenu", 0
   ...
.code
   ...
  LOCAL   hMenu: HMENU
   ...
  push    offset MenuName
  push    hInstance
  call    LoadMenu
  mov     hMenu, EAX
   ...
```

Полученный дескриптор hMenu можно указать в качестве девятого параметра функции CreateWindow:

```
invoke  CreateWindow, ... , hMenu, ...
mov     hWnd, EAX
```

Здесь необходимо отметить очень важный момент. При указании меню в вызове функции CreateWindow любое другое меню, заданное в классе окна, становится недоступным.

Можно также указать вместо меню NULL при регистрации класса окна и при вызове функции CreateWindow, а затем присоединить меню к окну при помощи вызова функции WIN API SetMenu. Функция SetMenu имеет следующий синтаксис:

```
BOOL  SetMenu(HWND     hWnd,              //дескриптор окна
              HMENU    hMenu              //дескриптор меню
);
```

Следующий фрагмент программного кода назначает окну hWnd меню с дескриптором hMenu:

```
  ...
push    hMenu
push    hWnd
call    SetMenu
  ...
```

Такая форма позволяет использовать несколько меню в одном приложении окна. Следует отметить, что любое меню, назначенное окну, удаляется при удалении окна. Не связанное с окном меню перед завершением работы программы нужно удалять при помощи вызова функции DestroyMenu. Поясню это на маленьком примере.

Пусть мы назначили окну с дескриптором hWnd меню, дескриптор которого находится в переменной hMenu1:

```
push    hMenu1
push    hWnd
call    SetMenu
```

Предположим, в программе нужно переопределить для окна приложения другое меню, например с дескриптором hMenu2. Это можно сделать, используя предыдущий фрагмент кода:

```
push    hMenu2
push    hWnd
call    SetMenu
```

Эти команды отсоединяют от окна ранее назначенное меню hMenu1, но не уничтожают его. При закрытии окна приложения будет уничтожено только меню hMenu2, а меню с дескриптором hMenu1 останется в памяти. Для удаления hMenu1 необходимо вызвать функцию DestroyMenu, принимающую в качестве параметра дескриптор меню:

```
DestroyMenu(hMenu1)
```

Теперь перейдем к рассмотрению основных примеров программ. В нашем первом примере будет присутствовать главное меню из двух пунктов и выпадающее меню, состоящее из трех пунктов. Это отображено в файле описания ресурсов, который назовем rsrc.rc:

```
#define IDM_WHITE    1
#define IDM_BLACK    2
#define IDM_BLUE     3
#define IDM_TITLE    4

MenuDemo MENU
{
  POPUP "&BACKGROUND"
  {
    MENUITEM "&WHITE BRUSH", IDM_WHITE
    MENUITEM "&BLACK BRUSH", IDM_BLACK
    MENUITEM  SEPARATOR
    MENUITEM "B&LUE BRUSH", IDM_BLUE
  }
  MENUITEM "&TITLE", IDM_TITLE
}
```

Каждый пункт главного и выпадающего меню описывается инструкцией MENUITEM с названием текстовой строки и идентификатора пункта меню.

При выборе пункта выпадающего меню цвет фона клиентской области окна будет меняться, а если выбран пункт главного меню TITLE, то будет изменен заголовок приложения. Исходный текст программы (назовем ее MENUEX) приведен в листинге 5.21.

Листинг 5.21. Программа, демонстрирующая работу с меню

```
;--------------------- MENUEX.ASM -----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib

  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------
  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szDisplayName DB "ДЕМОНСТРАЦИЯ РАБОТЫ С МЕНЮ", 0
  CommandLine   DD 0
  hWnd          DD 0
  hInstance     DD 0
  szClassName   DB "Demo_Class", 0

  AppName       DB "MyApp", 0
  MenuName      DB "MenuDemo", 0
  wColor        DB "ЦВЕТ ФОНА ПОМЕНЯЕТСЯ НА БЕЛЫЙ!", 0
  bkColor       DB "ЦВЕТ ФОНА ПОМЕНЯЕТСЯ НА ЧЕРНЫЙ!", 0
```

```
  blColor        DB "ЦВЕТ ФОНА ПОМЕНЯЕТСЯ НА ГОЛУБОЙ!", 0
  sTitle         DB "ЗАГОЛОВОК ОКНА ПОМЕНЯЕТСЯ!", 0

  toggle         DD 0
  sEng           DB "WINDOWS APPLICATION", 0
  sRus           DB "ПРИЛОЖЕНИЕ WINDOWS", 0
.const
  IDM_WHITE      EQU 1
  IDM_BLACK      EQU 2
  IDM_BLUE       EQU 3
  IDM_TITLE      EQU 4


  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain

  push    EAX
  call    ExitProcess

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD

  ; Локальные переменные процедуры

  LOCAL wc    :WNDCLASSEX
  LOCAL msg   :MSG
   ; Заполнение структуры WNDCLASSEX требуемыми параметрами
   mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW

  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL
```

```
push    hInst
pop     wc.hInstance
mov     wc.hbrBackground, COLOR_BTNFACE+1

mov     wc.lpszMenuName, offset MenuName
mov     wc.lpszClassName, offset szClassName

push    IDI_APPLICATION
push    NULL
call    LoadIcon
mov     wc.hIcon, EAX

push    IDC_ARROW
push    NULL
call    LoadCursor
mov     wc.hCursor, EAX
mov     wc.hIconSm, 0

lea     EAX, wc
push    EAX
call    RegisterClassEx

push    NULL
push    hInst
push    NULL
push    NULL

push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    CW_USEDEFAULT
push    WS_OVERLAPPEDWINDOW

push    offset szDisplayName
push    offset szClassName
push    WS_EX_OVERLAPPEDWINDOW
```

```
  call    CreateWindowEx
  mov     hWnd, EAX

  push    SW_SHOWNORMAL
  push    hWnd
  call    ShowWindow
   push    hWnd
  call    UpdateWindow
 ; Цикл обработки сообщений
 StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage
  cmp     EAX, 0
  je      ExitLoop
  lea     EAX, msg
  push    EAX
  call    TranslateMessage

  lea     EAX, msg
  push    EAX
  call    DispatchMessage

  jmp StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret

WinMain endp

; Оконная процедура

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
```

```
  LOCAL hdc     :HDC
  LOCAL ps      :PAINTSTRUCT
  LOCAL rect    :RECT
  LOCAL hBrush  :HBRUSH

  cmp     uMsg,WM_COMMAND
  jne     next_1
  mov     EAX, wParam
  cmp     AX, IDM_WHITE
  jne     check_black
  invoke  MessageBox, NULL, ADDR wColor, offset AppName, MB_OK
  mov     hBrush, COLOR_WINDOW+1
  jmp     fill_rect

check_black:
  cmp     AX,IDM_BLACK
  jne     check_blue
  invoke  MessageBox, NULL, ADDR bkColor, offset AppName, MB_OK
  mov     hBrush, COLOR_WINDOW+3
  jmp     fill_rect

check_blue:
  cmp     AX,IDM_BLUE
  jne     check_title
  invoke  MessageBox, NULL, ADDR blColor, offset AppName, MB_OK
  mov     hBrush, COLOR_WINDOW-3
  jmp     fill_rect
check_title:
  cmp     AX, IDM_TITLE
  jne     ex_wmsys
  invoke  MessageBox, NULL, ADDR sTitle, offset AppName, MB_OK
  cmp     toggle, 0
  je      chEng
  push    offset sRus
  push    hWin
  call    SetWindowText
  jmp     rev_flag
```

```
chEng:
  push    offset sEng
  push    hWin
  call    SetWindowText

rev_flag:
  not     toggle
ex_wmsys:
  ret

next_1:
  cmp     uMsg, WM_DESTROY
  jne     next_2
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret

next_2:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret

fill_rect:
  push    hWnd
  call    GetDC
  mov     hdc, EAX

  lea     ESI,rect
  push    ESI
  push    hWnd
  call    GetClientRect
   push    hBrush
  lea     ESI, rect
  push    ESI
```

```
  push    hdc
  call    FillRect
  push    hdc
  push    hWnd
  call    ReleaseDC
  ret

WndProc endp
```

Обратите внимание на раздел констант ассемблерной программы. Здесь объявлены идентификаторы нашего меню, причем их значения должны совпадать с определенными в файле описания ресурсов:

```
.const
  IDM_WHITE   EQU 1
  IDM_BLACK   EQU 2
  IDM_BLUE    EQU 3
  IDM_TITLE   EQU 4
```

При выборе пункта меню приложения Windows генерирует сообщение WM_COMMAND. При этом старшее слово переменной wParam равно 0, а младшее слово определяет идентификатор выбранного пункта меню. Например, фрагмент кода

```
  ...
cmp     uMsg, WM_COMMAND
jne     next_1
mov     EAX, wParam
cmp     AX, IDM_WHITE
  ...
```

анализирует выбор пункта меню IDM_WHITE. Обработчики пунктов IDM_WHITE, IDM_BLACK и IDM_BLUE выпадающего меню выполняют похожие действия — они меняют цвет фона клиентской области окна приложения. Например, при выборе пункта меню IDM_BLACK выводится предупреждающее сообщение, и цвет фона меняется на черный. Для изменения цвета мы используем функцию WIN API FillRect, имеющую следующий синтаксис:

```
int FillRect(HDC          hdc,    // дескриптор контекста устройства
             CONST RECT *lprc,    // указатель на структуру RECT,
```

```
                                   // определяющую область заполнения
              HBRUSH      hbr      // дескриптор логической кисти, которая
                                   // используется для заполнения области
              );
```

В качестве дескриптора кисти можно указать код системного цвета (COLOR_WINDOW) и добавить или вычесть из него определенное число, чтобы получить требуемый цвет. Заполнение клиентской области окна черным цветом выполняется следующим кодом:

```
   ...
  mov     hBrush, COLOR_WINDOW+3
   ...
fill_rect:
   ...
  push    hBrush
  lea     ESI, rect
  push    ESI
  push    hdc
  call    FillRect
   ...
```

Выбор пункта меню TITLE приводит к выполнению программного кода, меняющего заголовок окна приложения. Здесь используется функция SetWindowText, принимающая в качестве параметров дескриптор окна и адрес строки текста. Переменная toggle служит для переключения с одного заголовка на другой.

Окно работающего приложения изображено на рис. 5.15.



Рис. 5.15. Окно приложения, демонстрирующего работу стандартного меню

В следующем примере показан способ загрузки меню при помощи функции SetMenu. Такой вариант удобен при необходимости оперировать с несколькими меню в программе. Смена меню выполняется при нажатии кнопок мыши. Исходный текст программы (назовем ее DMENU) приведен далее в листинге 5.22.

Листинг 5.22. Программа, демонстрирующая работу с несколькими меню

```
;---------------------- DMENU.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ;--------
  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
  DrawTxt PROTO :DWORD,:DWORD,:DWORD

.data
  szDisplayName DB "ДИНАМИЧЕСКИЕ МЕНЮ", 0
  CommandLine   DD 0
  hWnd          DD 0
  hInstance     DD 0
  szClassName   DB "Demo_Class", 0

  AppName       DB "МОЯ ПРОГРАММА", 0

  ;---- Установки для меню 1 ----

  MenuName1     DB "MenuDemo1", 0
```

```
  Text11        DB "ВЫ ВЫБРАЛИ ОПЦИЮ TEXT11", 0
  Text12        DB "ВЫ ВЫБРАЛИ ОПЦИЮ TEXT12", 0
  Ex1           DB "ВЫ ВЫБРАЛИ ОПЦИЮ EXIT1", 0
  Mes1          DB "ВЫ ВЫБРАЛИ ПУНКТ MESSAGE1", 0
  ;---- Установки для меню 2 ----

  MenuName2     DB "MenuDemo2", 0
  Text21        DB "ВЫ ВЫБРАЛИ ОПЦИЮ TEXT12", 0
  Text22        DB "ВЫ ВЫБРАЛИ ОПЦИЮ TEXT22", 0
  Ex2           DB "ВЫ ВЫБРАЛИ ОПЦИЮ EXIT2", 0
  Mes2          DB "ВЫ ВЫБРАЛИ ПУНКТ MESSAGE2", 0

.const

  IDM_TEXT11    EQU 1
  IDM_TEXT12    EQU 2
  IDM_EXIT1     EQU 3
  IDM_MESSAGE1  EQU 4

  IDM_TEXT21    EQU 5
  IDM_TEXT22    EQU 6
  IDM_EXIT2     EQU 7
  IDM_MESSAGE2  EQU 8

.code
start:
  push    NULL
  call    GetModuleHandle
  mov     hInstance, EAX

  call    GetCommandLine
  mov     CommandLine, EAX

  push    SW_SHOWDEFAULT
  push    CommandLine
  push    NULL
  push    hInstance
  call    WinMain
```

```
 push    EAX
 call    ExitProcess
WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
  ; Локальные переменные процедуры

 LOCAL wc    :WNDCLASSEX
 LOCAL msg   :MSG

 ; Заполнение структуры WNDCLASSEX требуемыми параметрами

 mov     wc.cbSize, sizeof WNDCLASSEX
 mov     wc.style, CS_HREDRAW or CS_VREDRAW

 mov     wc.lpfnWndProc, offset WndProc
 mov     wc.cbClsExtra, NULL
 mov     wc.cbWndExtra, NULL

 push    hInst
 pop     wc.hInstance

 mov     wc.hbrBackground, COLOR_WINDOW-2
 mov     wc.lpszMenuName, NULL
 mov     wc.lpszClassName, offset szClassName

 push    IDI_APPLICATION
 push    NULL
 call    LoadIcon
 mov     wc.hIcon, EAX

 push    IDC_ARROW
 push    NULL
 call    LoadCursor
 mov     wc.hCursor, EAX
 mov     wc.hIconSm, 0
```

```
  lea     EAX, wc
  push    EAX
  call    RegisterClassEx

  push    NULL
  push    hInst
  push    NULL
  push    NULL

  push    CW_USEDEFAULT
  push    CW_USEDEFAULT
  push    CW_USEDEFAULT
  push    CW_USEDEFAULT

  push    WS_OVERLAPPEDWINDOW
  push    offset szDisplayName
  push    offset szClassName
  push    WS_EX_OVERLAPPEDWINDOW
  call    CreateWindowEx
  mov     hWnd, EAX

  push    SW_SHOWNORMAL
  push    hWnd
  call    ShowWindow

  push    hWnd
  call    UpdateWindow

; Цикл обработки сообщений

StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage
   cmp     EAX, 0
```

```
  je       ExitLoop

  lea      EAX, msg
  push     EAX
  call     TranslateMessage

  lea      EAX, msg
  push     EAX
  call     DispatchMessage
   jmp      StartLoop

ExitLoop:
  mov      EAX, msg.wParam
  ret

WinMain endp

; Оконная процедура

WndProc proc hWin    :DWORD,
             uMsg    :DWORD,
             wParam  :DWORD,
             lParam  :DWORD

  ; Локальные переменные

  LOCAL hdc    :HDC
  LOCAL ps     :PAINTSTRUCT
  LOCAL rect   :RECT
  LOCAL coord  :DWORD
  LOCAL hMenu  :HMENU
   cmp      uMsg,WM_COMMAND
  jne      next_1
  mov      EAX, wParam
  cmp      AX, IDM_TEXT11
  jne      check_text12
  invoke   MessageBox, NULL, ADDR Text11, offset AppName, MB_OK
  jmp      ex_wmsys
```

```
check_text12:
  cmp      AX,IDM_TEXT12
  jne      check_ex1
  invoke   MessageBox, NULL, ADDR Text12, offset AppName, MB_OK
  jmp      ex_wmsys

check_ex1:
  cmp      AX,IDM_EXIT1
  jne      check_mes1
  invoke   MessageBox, NULL, ADDR Ex1, offset AppName, MB_OK
  jmp      ex_wmsys

check_mes1:
  cmp      AX, IDM_MESSAGE1
  jne      next_menu
  invoke   MessageBox, hWin, ADDR Mes1, offset AppName, MB_OK
  jmp      ex_wmsys

next_menu:
  cmp      AX,IDM_TEXT21
  jne      check_text22
  invoke   MessageBox, NULL, ADDR Text21, offset AppName, MB_OK
  jmp      ex_wmsys

check_text22:
  cmp      AX,IDM_TEXT22
  jne      check_ex2
  invoke   MessageBox, NULL, ADDR Text22, offset AppName, MB_OK
  jmp      ex_wmsys

check_ex2:
  cmp      AX,IDM_EXIT2
  jne      check_mes2
  invoke   MessageBox, NULL, ADDR Ex2, offset AppName, MB_OK
  jmp      ex_wmsys

check_mes2:
  cmp      AX, IDM_MESSAGE2
```

```
  jne      ex_wmsys
  invoke   MessageBox, hWin, ADDR Mes2, offset AppName, MB_OK
ex_wmsys:   ret

next_1:
  cmp      uMsg, WM_LBUTTONDOWN
  jne      next_2
  invoke   LoadMenu, hInstance, ADDR MenuName2
  mov      hMenu, EAX
  invoke   SetMenu, hWin, hMenu
  ret
 next_2:
  cmp      uMsg, WM_RBUTTONDOWN
  jne      next_3
  invoke   LoadMenu, hInstance, ADDR MenuName1
  mov      hMenu, EAX
  invoke   SetMenu, hWin, hMenu
  ret

next_3:
  cmp      uMsg, WM_DESTROY
  jne      next_4
  push     NULL
  call     PostQuitMessage
  xor      EAX, EAX
  ret

next_4:
  push     lParam
  push     wParam
  push     uMsg
  push     hWin
  call     DefWindowProc
  ret
 WndProc endp

end start
```

Содержимое файла описания ресурсов этого приложения должно быть следующим:

```
#define IDM_TEXT11    1
#define IDM_TEXT12    2
#define IDM_EXIT1     3

#define IDM_MESSAGE1  4
#define IDM_TEXT21    5

#define IDM_TEXT22    6
#define IDM_EXIT2     7
#define IDM_MESSAGE2  8

MenuDemo1 MENU
{
  POPUP "&TEXT1"
  {
    MENUITEM "&TEXT11", IDM_TEXT11
    MENUITEM "T&EXT12", IDM_TEXT12

    MENUITEM  SEPARATOR
    MENUITEM "E&xit1", IDM_EXIT1
  }
  MENUITEM "&MESSAGE1", IDM_MESSAGE1
}

MenuDemo2 MENU
{
  POPUP "&TEXT2"
  {
    MENUITEM "&TEXT21", IDM_TEXT21
    MENUITEM "T&EXT22", IDM_TEXT22

    MENUITEM  SEPARATOR
    MENUITEM "E&xit2", IDM_EXIT2
  }
  MENUITEM "&MESSAGE2", IDM_MESSAGE2
}
```

Окно работающего приложения изображено на рис. 5.16.



Рис. 5.16. Окно приложения с переключаемыми меню

5.8. Использование элементов управления

Операционная система Windows имеет предопределенные классы окон, которые можно использовать при написании приложений. Такими классами являются кнопки, списки, поля со списком, поля редактирования, линейки прокрутки и др. Созданные на их основе элементы управления придают дополнительную гибкость и мощь интерфейсу приложения. Элементы управления представляют собой дочерние окна и создаются в приложении при помощи функций WIN API `CreateWindow` или `CreateWindowEx`. Регистрировать классы окон, как это делается для главного окна приложения, не нужно, т. к. эти классы уже зарегистрированы в Windows.

К существующим классам, с которыми может работать функция `CreateWindow`, относятся `Button`, `ListBox`, `ComboBox`, `Static`, `RichEdit` и `ScrollBar`. Есть некоторые особенности работы с элементами управления, созданными на базе этих классов. Предположим, вы создаете в программе элемент управления "кнопка". Для этого в функции `CreateWindow` вы должны указать в имени класса `Button`, а также дескриптор родительского окна и идентификатор создаваемой кнопки.

Обычно элементы управления создаются при вызове обработчика сообщения `WM_CREATE` главного окна приложения. Дочернее окно, которым является элемент управления, может посылать сообщение `WM_COMMAND` родительскому окну. При этом идентификатор `ID` элемента управления помещается в младшее слово переменной `wParam`, а в старшее слово помещается код, уведомляющий о том, что с элементом управления произошло. Например, если уведомляющий код для кнопки имеет значение `BN_CLICKED`, это говорит о том, что кнопка нажата.

Сейчас мы создадим программу, в которой выполним преобразование текстовой строки. В первое поле редактирования (элемент управления `Edit`) введем текст, затем заменим во введенной строке все пробелы на дефис, после чего выведем результат в окно второго поля редактирования. Преобразование будет выполняться при нажатии на кнопку (элемент управления `Button`).

Кроме этого, разместим над полями редактирования подсказки, использовав два элемента статического текста `Static`. Исходный текст программы (назовем ее CONTR) приведен в листинге 5.23.

Листинг 5.23. Программа, демонстрирующая работу основных элементов управления Windows

```
;---------------------- CONTR.ASM -----------------------
.386
.model flat,stdcall
  option casemap:none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib

  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  ClassName         DB "SimpleWinClass", 0
  AppName           DB " ", 0
  CommandLine       DD 0
  hInstance         DD 0

  ; Переменные для элемента "button"

  ButtonClassName   DB "BUTTON", 0
  ButtonText        DB "ПРЕОБРАЗОВАТЬ", 0
  hwndButton        DD 0
```

```
; Переменные для элемента "STATIC"

LabelClassName     DB "STATIC", 0
LabelText1         DB "  ИСХОДНЫЙ ТЕКСТ", 0
LabelText2         DB "  ПРЕОБРАЗОВАННЫЙ ТЕКСТ", 0
hwndLabel1         DD 0
hwndLabel2         DD 0

; Переменные для элемента "EDIT"

EditClassName      DB "edit", 0
hwndEdit1          DD 0
hwndEdit2          DD 0
bytesWritten       DD 0
buffer             DB 512 dup(?)

; Идентификаторы элементов управления

.const
  ButtonID         EQU 1
  EditID1          EQU
  EditID2          EQU
  LabelID1         EQU 4
  LabelID2         EQU 5

.code
start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX
  invoke  GetCommandLine
  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, EAX

WinMain proc hInst:HINSTANCE,
             hPrevInst:HINSTANCE,
             CmdLine:LPSTR,
             CmdShow:DWORD
```

```
  LOCAL wc    :WNDCLASSEX
  LOCAL msg   :MSG
  LOCAL hwnd  :HWND

  mov     wc.cbSize, SIZEOF WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW
  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL
  push    hInst
  pop     wc.hInstance
  mov     wc.hbrBackground,COLOR_WINDOW-2
  mov     wc.lpszMenuName, NULL

  mov     wc.lpszClassName, offset ClassName
  invoke  LoadIcon, NULL, IDI_APPLICATION
  mov     wc.hIcon, EAX
  mov     wc.hIconSm, EAX
  invoke  LoadCursor, NULL, IDC_ARROW
  mov     wc.hCursor, EAX
  invoke  RegisterClassEx, addr wc

  invoke  CreateWindowEx, WS_EX_CLIENTEDGE, ADDR ClassName,\
                          ADDR AppName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL,\
                          hInst, NULL
  mov     hwnd, EAX

  invoke  ShowWindow, hwnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hwnd

; Цикл обработки сообщений

StartLoop:
  invoke  GetMessage, ADDR msg, NULL, 0, 0
  cmp     EAX, 0
  je      ExitLoop
```

```
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp

; Оконная процедура

WndProc proc hWin:HWND,
             uMsg:UINT,
             wParam:WPARAM,
             lParam:LPARAM

  cmp     uMsg, WM_CREATE
  jne     next_1
  invoke  CreateWindowEx, WS_EX_CLIENTEDGE,\
                          ADDR LabelClassName, NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER\
                          or ES_LEFT or ES_AUTOHSCROLL,\
                          20, 35, 170, 25, hWin, LabelID1,\
                          hInstance, NULL
  mov     hwndLabel1, EAX

  invoke  CreateWindowEx, WS_EX_CLIENTEDGE,\
                          ADDR LabelClassName, NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER\
                          or ES_LEFT or ES_AUTOHSCROLL,\
                          200, 35, 220, 25, hWin, LabelID2,\
                          hInstance, NULL
  mov     hwndLabel2, EAX

  invoke  CreateWindowEx, WS_EX_CLIENTEDGE,\
                          ADDR EditClassName, NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER\
                          or ES_LEFT or ES_AUTOHSCROLL,\
                          20, 65, 170, 25, hWin, EditID1,\
                          hInstance, NULL
```

```
  mov     hwndEdit1, EAX

  invoke  CreateWindowEx, WS_EX_CLIENTEDGE,\
                          ADDR EditClassName, NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER\
                          or ES_LEFT or ES_AUTOHSCROLL,\
                          200, 65, 220, 25, hWin, EditID2,\
                          hInstance, NULL
  mov     hwndEdit2, EAX

  invoke  CreateWindowEx, NULL, ADDR ButtonClassName, ADDR ButtonText,\
                          WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,\
                          140, 95, 130, 25, hWin, ButtonID,\
                          hInstance, NULL
  mov     hwndButton, EAX

  invoke  SetFocus, hwndEdit1
  invoke  SetWindowText, hwndLabel1, ADDR LabelText1
  invoke  SetWindowText, hwndLabel2, ADDR LabelText2
  ret

next_1:
  cmp     uMsg, WM_COMMAND
  jne     next_2
  mov     EAX,wParam
  cmp     lParam, 0
  je      ex_wmcom
  cmp     AX, ButtonID

  jne     ex_wmcom
  shr     EAX, 16
  cmp     AX, BN_CLICKED
  jne     ex_wmcom

  invoke  GetWindowText, hwndEdit1, ADDR buffer, 512
  mov     bytesWritten, EAX
  call    ReplaceSpace
  invoke  SetWindowText, hwndEdit2, ADDR buffer
```

```
ex_wmcom:
  ret
next_2:
  cmp     uMsg, WM_DESTROY
  jne     next_3
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret

next_3:
  push    lParam
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret
WndProc endp

ReplaceSpace proc
  lea     EDI, buffer
  mov     ECX, bytesWritten
  cld
  mov     AL, 20h
next_ch:
  scasb
  je      repSpace
cont:
  loop    next_ch
  ret
repSpace:
  mov     byte ptr [EDI-1], '-'
  jmp     cont
ReplaceSpace endp

end start
```

Анализ программного кода начнем с рассмотрения процесса создания элементов управления. Все они создаются при помощи функции `CreateWindowEx` по схожему сценарию, поэтому достаточно подробно остановиться на только одном элементе, например кнопке. Кнопка создается при выполнении следующего фрагмента кода:

```
invoke  CreateWindowEx, NULL, ADDR ButtonClassName, ADDR ButtonText,\
                        WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,\
                        140, 95, 130, 25, hWin, ButtonID, hInstance, NULL
mov     hwndButton, EAX
```

Среди параметров вызова функции следует выделить:

- `ADDR ButtonClassName` — указатель на строку с именем класса (`Button`);
- `WS_CHILD` — информирует о том, что создаваемое окно является дочерним;
- `HWin` — дескриптор родительского окна;
- `ButtonID` — идентификатор элемента управления.

Дескриптор вновь созданного окна сохраняем в переменной `hwndButton`. При нажатии на кнопку приложению посылается сообщение `WM_COMMAND`. Оконная процедура анализирует это сообщение, как показано в следующем фрагменте программного кода:

```
 ...
cmp     uMsg, WM_COMMAND
jne     next_2
mov     EAX,wParam
cmp     lParam, 0
je      ex_wmcom
cmp     AX, ButtonID

jne     ex_wmcom
shr     EAX, 16
cmp     AX, BN_CLICKED
jne     ex_wmcom
invoke  GetWindowText, hwndEdit1, ADDR buffer, 512
mov     bytesWritten, EAX
```

```
  call    ReplaceSpace
  invoke  SetWindowText, hwndEdit2, ADDR buffer
ex_wmcom:
  ret
   ...
```

Вначале анализируем, содержит ли параметр `lParam` значение, отличное от 0. Если да, то сообщение инициировано элементом управления, и мы идем дальше. Значение `lParam`, равное 0, говорит о том, что сообщение вызвано, например, выбором пункта меню, что нас не интересует.

```
  ...
mov     EAX,wParam
cmp     lParam, 0
je      ex_wmcom
  ...
```

Предположим, что сообщение вызвано элементом управления, т. е. переменная `lParam` отлична от 0. Дальше необходимо определить, кнопка ли это. Здесь нам пригодится идентификатор кнопки `ButtonID`, определенный ранее:

```
ButtonID          EQU 1
```

Вспомним, что в регистре `EAX` к этому моменту находится значение `wParam`, причем регистр `AX` содержит идентификатор элемента управления (`lParam` не равен 0). Следующий фрагмент кода определяет, является ли элемент управления кнопкой и, если да, то была ли кнопка нажата (`BN_CLICKED`):

```
  cmp     AX, ButtonID
  jne     ex_wmcom             ; нет, это не кнопка.Выходим из обработчика
  shr     EAX, 16              ; если это кнопка, то сдвигаем старшие биты
                               ; в AX для последующего анализа
  cmp     AX, BN_CLICKED
  jne     ex_wmcom             ; если кнопка нажата, обрабатываем нажатие

  invoke  GetWindowText, hwndEdit1, ADDR buffer, 512
  mov     bytesWritten, EAX
  call    ReplaceSpace
```

```
  invoke  SetWindowText, hwndEdit2, ADDR buffer
ex_wmcom:
  ret
```

Текст из поля редактирования с дескриптором hwndEdit1 копируется в буфер с максимальным размером в 512 байт при помощи функции WIN API GetWindowText. В качестве результата функция возвращает количество байт, фактически скопированных в буфер. Это значение сохраняется в переменной bytesWritten. Далее вызываем процедуру ReplaceSpace, которая использует адрес буфера и число скопированных байт в качестве параметров. После преобразования текст выводится из буфера памяти в окно второго поля редактирования (дескриптор hwndEdit2).

Окно работающего приложения изображено на рис. 5.17.

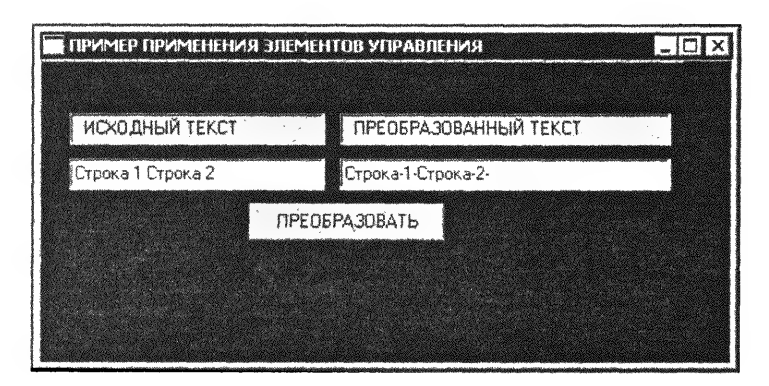


Рис. 5.17. Окно приложения, выполняющего преобразование текста

5.9. Диалоговые окна и их использование

Диалоговое окно во многом напоминает обычное выпадающее окно Windows. Основное различие между ними состоит в том, что в диалоговых окнах применяются шаблоны, определяющие элементы управления, создаваемые в них. Все такие шаблоны определяются в файле описания ресурса для типа DIALOGEX. Можно сказать и так: диалоговое окно — это обычное окно Windows, которое разработано с дочерними окнами (элементами управления) для дальнейшего использования. Кроме того, операционная система снабжает диалоговое окно возможностями обработки клавиатурного ввода, например нажатий клавиш сдвига, <Tab> и <Enter>, чего нет в обычном окне.

Окно диалога определяется в файле описания ресурсов. Можно написать шаблон диалогового окна вместе с элементами управления в нем, затем от-

компилировать его при помощи редактора ресурсов. Вот как может выглядеть содержимое файла ресурсов (назовем его rsrc.rc) для нашего следующего примера:

```
#define IDC_EDIT1        3000
#define IDC_EDIT2        3001
#define IDC_EDIT3        3002
#define IDC_PLUS         3003
#define IDC_CLEAR        3004

#define DS_CENTER        0x0800L
#define WS_CAPTION       0x00C00000L
#define WS_VISIBLE       0x10000000L
#define WS_SYSMENU       0x00080000L
#define WS_MINIMIZEBOX   0x00020000L
#define WS_VISIBLE       0x10000000L
#define WS_OVERLAPPED    0x00000000L

#define DS_MODALFRAME    0x80L
#define DS_3DLOOK        0x0004L
#define WS_TABSTOP       0x00010000L
#define ES_AUTOHSCROLL   0x0080L
#define ES_LEFT          0x0000L

MyDialog DIALOG 10, 10, 300, 100
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_MINIMIZEBOX |
WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK

CAPTION "ИСПОЛЬЗОВАНИЕ ОКНА ДИАЛОГА ДЛЯ СУММЫ ДВУХ ЧИСЕЛ (ВАР.1)"
CLASS "DIALOGEX"
BEGIN

  EDITTEXT  IDC_EDIT1, 75,17,40,13, ES_AUTOHSCROLL | ES_LEFT |WS_TABSTOP
  EDITTEXT  IDC_EDIT2, 75,37,40,13, ES_AUTOHSCROLL | ES_LEFT |WS_TABSTOP
  EDITTEXT  IDC_EDIT3, 75,57,40,13, ES_AUTOHSCROLL | ES_LEFT |WS_TABSTOP
  DEFPUSHBUTTON   " + ", IDC_PLUS,    141,17,52,13
  PUSHBUTTON      "&Clear", IDC_CLEAR,141,37,52,13

END
```

В этом скрипте определяется диалоговое окно с размещенными на нем элементами управления — тремя полями редактирования для ввода-вывода текста и двумя кнопками. Наша программа будет складывать два целых числа и результат выводить в поле редактирования. Исходный текст приложения (назовем его DIALOG1) приведен в листинге 5.24.

Листинг 5.24. Программа, демонстрирующая работу с диалоговым окном

```
;-------------------- DIALOG1.ASM ----------------------
.386
.model flat,stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib

  WinMain proto :DWORD,:DWORD,:DWORD,:DWORD

.data
  ClassName        DB "DIALOGEX", 0
  DlgName          DB "MyDialog", 0
  AppName          DB " Add two integers", 0

  hInstance        DD 0
  CommandLine      DD 0
  buffer           DB 512 dup(?)
  i1               DD 0
  i2               DD 0
  ires             DD 0
  lpTranslated     DD 0

  LabelClassName   DB "STATIC", 0
  LabelText1       DB "1-Е СЛАГАЕМОЕ", 0
  LabelText2       DB "2-Е СЛАГАЕМОЕ", 0
  LabelText3       DB "СУММА", 0
  hwndLabel1       DD 0
```

```
  hwndLabel2       DD 0
  hwndLabel3       DD 0

.const

  IDC_EDIT1        EQU 3000
  IDC_EDIT2        EQU 3001
  IDC_EDIT3        EQU 3002
  IDC_PLUS         EQU 3003
  IDC_CLEAR        EQU 3004

  LabelID1         EQU 3005
  LabelID2         EQU 3006
  LabelID3         EQU 3007

.code

start:

  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX
  invoke  GetCommandLine
  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, EAX
 WinMain proc hInst      :HINSTANCE,
              HPrevInst :HINSTANCE,
              CmdLine   :LPSTR,
              CmdShow   :DWORD

  LOCAL wc    :WNDCLASSEX
  LOCAL msg   :MSG
  LOCAL hDlg  :HWND

  mov     wc.cbSize,SIZEOF WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW
  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, DLGWINDOWEXTRA
```

```
  push    hInst
  pop     wc.hInstance
  mov     wc.hbrBackground, COLOR_WINDOW-3
 mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset ClassName

  invoke  LoadIcon, NULL, IDI_APPLICATION
  mov     wc.hIcon, EAX
  mov     wc.hIconSm, EAX
  invoke  LoadCursor, NULL, IDC_ARROW
  mov     wc.hCursor, EAX
  invoke  RegisterClassEx, addr wc

  invoke  CreateDialogParam, hInstance, ADDR DlgName, NULL, NULL, NULL
  mov     hDlg, EAX
  invoke  GetDlgItem, hDlg, IDC_EDIT1
  invoke  SetFocus, EAX
  invoke  ShowWindow, hDlg,SW_SHOWNORMAL
  invoke  UpdateWindow, hDlg

StartLoop:
  invoke  GetMessage, ADDR msg, NULL, 0, 0
  cmp     EAX, 0
  je      ExitLoop
  invoke  IsDialogMessage, hDlg, ADDR msg
  cmp     EAX, 0
  jne     StartLoop
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret

WinMain endp

; Оконная процедура
```

```
WndProc proc hWin    :HWND,
             UMsg    :UINT,
             wParam  :WPARAM,
             lParam  :LPARAM

  cmp     uMsg, WM_CREATE
  jne     next_1
  invoke  CreateWindowEx, WS_EX_CLIENTEDGE, ADDR LabelClassName, NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER or\
                          ES_LEFT or ES_AUTOHSCROLL,\
                          15, 35, 130, 25, hWin,\
                          LabelID1, hInstance, NULL
  mov     hwndLabel1, EAX

  invoke  CreateWindowEx, WS_EX_CLIENTEDGE, ADDR LabelClassName, NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER or\
                          ES_LEFT or ES_AUTOHSCROLL,\
                          15, 75, 130, 25, hWin,\
                          LabelID2, hInstance, NULL

  mov     hwndLabel2, EAX
  invoke  CreateWindowEx, WS_EX_CLIENTEDGE, ADDR LabelClassName, NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER or\
                          ES_LEFT or ES_AUTOHSCROLL,\
                          15, 115, 130, 25, hWin,\
                          LabelID3, hInstance, NULL

  mov     hwndLabel3, EAX
  invoke  SetWindowText, hwndLabel1, ADDR LabelText1
  invoke  SetWindowText, hwndLabel2, ADDR LabelText2
  invoke  SetWindowText, hwndLabel3, ADDR LabelText3
  ret

next_1:
  cmp     uMsg, WM_COMMAND
  jne     next_2
  mov     EAX, wParam
  mov     EDX, EAX
```

```
  cmp     lParam, 0
  je      ex_wmcom
  shr     EDX, 16
  cmp     DX, BN_CLICKED
  je      check_plus
  jmp     ex_wmcom
check_plus:
  cmp     AX, IDC_PLUS
  jne     check_clear
  invoke  GetDlgItemInt, hWin, IDC_EDIT1, ADDR lpTranslated, 1
  mov     i1, EAX
  invoke  GetDlgItemInt, hWin, IDC_EDIT2, ADDR lpTranslated, 1
  add     i1, EAX
  xchg    i1, EAX
  mov     ires, EAX
  invoke  SetDlgItemInt, hWin, IDC_EDIT3, ires, 1
  ret

  jmp     ex_wmcom
check_clear:
  cmp     AX, IDC_CLEAR
  jne     ex_wmcom

  invoke  SetDlgItemText, hWin, IDC_EDIT1, NULL
  invoke  SetDlgItemText, hWin, IDC_EDIT2, NULL
  invoke  SetDlgItemText, hWin, IDC_EDIT3, NULL

ex_wmcom:
  ret
next_2:
  cmp     uMsg, WM_DESTROY
  jne     next_3
  push    NULL
  call    PostQuitMessage
  xor     EAX, EAX
  ret
next_3:
  push    lParam
```

```
  push    wParam
  push    uMsg
  push    hWin
  call    DefWindowProc
  ret

WndProc endp

end start
```

Анализ примера начнем с определения диалогового окна в файле ресурсов. Строка:

```
MyDialog DIALOG 10, 10, 300, 100
```

указавает на координаты диалогового окна. Строка:

```
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_MINIMIZEBOX |
WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
```

определяет стили диалогового окна. Строка:

```
CAPTION "ИСПОЛЬЗОВАНИЕ ОКНА ДИАЛОГА ДЛЯ СУММЫ ДВУХ ЧИСЕЛ (ВАР.1)"
```

представляет собой заголовок окна. Наконец, блок:

```
CLASS "DIALOGEX"
BEGIN
  EDITTEXT  IDC_EDIT1, 75,17,40,13, ES_AUTOHSCROLL | ES_LEFT |WS_TABSTOP
  EDITTEXT  IDC_EDIT2, 75,37,40,13, ES_AUTOHSCROLL | ES_LEFT |WS_TABSTOP
  EDITTEXT  IDC_EDIT3, 75,57,40,13, ES_AUTOHSCROLL | ES_LEFT |WS_TABSTOP
  DEFPUSHBUTTON    " + ", IDC_PLUS,    141,17,52,13
  PUSHBUTTON       "&Clear", IDC_CLEAR,141,37,52,13
END
```

определяет элементы управления диалогового окна.

Обратимся теперь к исходному тексту программы. Диалоговое окно представляет собой одну из разновидностей окон Windows, поэтому регистрация класса окна и создание экземпляра окна являются стандартными и не отли-

чаются в принципе от аналогичных процедур для других окон. Разница лишь в том, что все эти манипуляции выполняются при помощи специальных функций, предназначенных для этого. Команда:

```
mov     wc.lpszClassName, offset ClassName
```

заполняет поле структуры wc именем класса. Для создания и отображения самого окна используется функция CreateDialogParam:

```
invoke  CreateDialogParam, hInstance, ADDR DlgName, NULL, NULL, NULL
mov     hDlg,EAX
```

Полученный дескриптор окна запоминается в переменной hDlg. Кроме самого окна диалога у нас появятся еще 8 элементов управления: три элемента статического текста, три поля редактирования и две кнопки. Все такие элементы создаются с помощью все той же функции CreateWindow. Например, для создания элемента статического текста необходимо выполнить следующие команды:

```
invoke  CreateWindowEx, WS_EX_CLIENTEDGE, ADDR LabelClassName, NULL,\
                        WS_CHILD or WS_VISIBLE or WS_BORDER or\
                        ES_LEFT or ES_AUTOHSCROLL,\
                        15, 35, 130, 25, hWin,\
                        LabelID1, hInstance, NULL
mov     hwndLabel1, EAX
```

Создание элементов управления выполняют обычно в обработчике WM_CREATE. Для управления поведением элемента управления, расположенного в диалоговом окне, вызывается функция GetDlgItem. Кроме того, в Windows имеется целый ряд функций для работы с элементами управления в диалоговом окне. В нашем примере мы использовали функции WIN API GetDlgItemInt, SetDlgItemInt, SetDlgItemText.

Функция GetDlgItemInt представляет собой сокращенный метод выборки целочисленного значения из элемента управления в диалоговом окне. Эта функция преобразует символьную строку в элементе управления в целое число и имеет синтаксис:

```
UINT GetDlgItemInt(HWND  hDlg,        // дескриптор диалогового окна
                   int   nIDDlgItem, // идентификатор элемента управления
```

```
                 BOOL *lpTranslated,  // указатель на переменную,
                                   // определяющую, успешно ли
                                   // выполнено преобразование
                 BOOL  bSigned     // TRUE, если выбираемое значение –
                                   // целое число со знаком,
                                   // FALSE, если целое число без знака
                 );
```

Функция `SetDlgItemInt` позволяет установить текстовое представление целого числа и объявляется так:

```
BOOL SetDlgItemInt(HWND  hDlg,       // дескриптор диалогового окна
                   int   nIDDlgItem, // идентификатор элемента управления
                   UINT  uValue,     // целочисленное значение, которое
                                  // необходимо установить в качестве
                                  // текста в элементе управления
                   BOOL  bSigned  // то же, что и в функции GetDlgItemInt
                   );
```

Наконец, с помощью функции `SetDlgItemText` можно установить заголовок элемента управления. Функция имеет синтаксис:

```
BOOL SetDlgItemText(HWND  hDlg,        // дескриптор диалогового окна
                    int   nIDDlgItem,  // идентификатор элемента
                                       // управления
                    LPCTSTR lpString   //буфер текста
                    );
```

Наша программа представляет собой простейший калькулятор, выполняющий всего одну операцию — сложение целых чисел.

Сообщения от элементов управления отрабатываются в обработчике `WM_COMMAND`. Для анализа приходящих вместе с этим сообщением параметров `wParam` и `lParam` разработан следующий фрагмент программного кода:

```
next_1:
  cmp     uMsg, WM_COMMAND
  jne     next_2
  mov     EAX, wParam
```

```
  mov     EDX, EAX
  cmp     lParam, 0
  je      ex_wmcom
```

Для того чтобы получить сообщение от конкретного элемента управления, необходимо проанализировать параметр lParam. Мы уже встречались с этим в предыдущем примере. Если lParam не равен 0, то программа определяет, какой элемент управления вызвал сообщение и реагирует соответствующим образом.

Например, при нажатии кнопки "+" инициируется сообщение IDC_PLUS и происходит сложение двух чисел:

```
   ...
check_plus:
  cmp     AX, IDC_PLUS
  jne     check_clear
  invoke  GetDlgItemInt, hWin, IDC_EDIT1, ADDR lpTranslated, 1
  mov     i1, EAX
  invoke  GetDlgItemInt, hWin, IDC_EDIT2, ADDR lpTranslated, 1
  add     i1, EAX
  xchg    i1, EAX
  mov     ires, EAX
  invoke  SetDlgItemInt, hWin, IDC_EDIT3, ires, 1
   ...
```

При нажатии кнопки Clear все поля редактирования очищаются:

```
   ...
check_clear:
  cmp     AX, IDC_CLEAR
  jne     ex_wmcom
  invoke  SetDlgItemText, hWin, IDC_EDIT1, NULL
  invoke  SetDlgItemText, hWin, IDC_EDIT2, NULL
  invoke  SetDlgItemText, hWin, IDC_EDIT3, NULL
   ...
check_clear:
  cmp     AX, IDC_CLEAR
  jne     ex_wmcom
```

```
  invoke  SetDlgItemText, hWin, IDC_EDIT1, NULL
  invoke  SetDlgItemText, hWin, IDC_EDIT2, NULL
  invoke  SetDlgItemText, hWin, IDC_EDIT3, NULL
```

В этом примере также показано, как можно вывести текст в элемент управления. Это осуществляется с помощью следующего фрагмента кода:

```
  ...
mov      hwndLabel3, EAX
invoke   SetWindowText, hwndLabel1, ADDR LabelText1

invoke   SetWindowText, hwndLabel2, ADDR LabelText2
invoke   SetWindowText, hwndLabel3, ADDR LabelText3
  ...
```

Окно работающего приложения изображено на рис. 5.18.

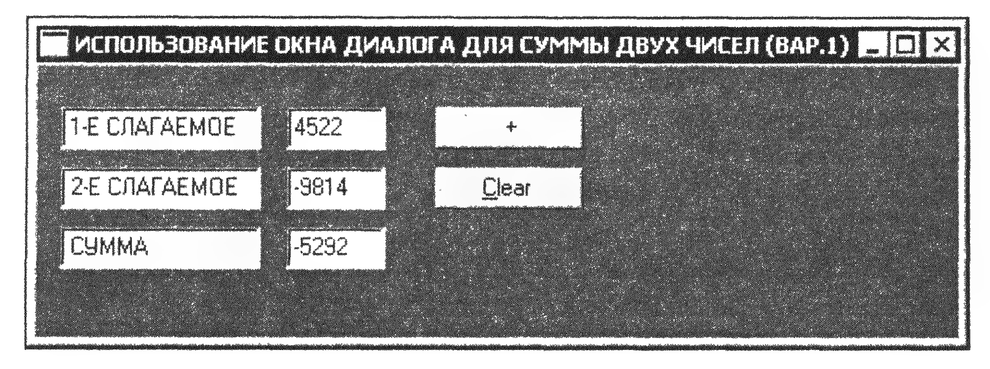


Рис. 5.18. Окно приложения, демонстрирующего работу стандартных элементов управления Windows

5.10. Применение библиотек динамической компоновки (DLL)

Библиотеки динамической компоновки (Dynamic Link Libraries — DLL) являются неотъемлемой и, пожалуй, наиболее важной частью операционных систем Windows. Они служат хранилищем многочисленных процедур, в том числе и функций WIN API, и являются мощным средством для написания эффективных приложений. Не будем останавливаться на принципах построения и функционирования DLL, поскольку имеется масса публикаций,

посвященных этой теме. Гораздо более интересно научиться самому создавать библиотеки динамической компоновки. Библиотеки DLL, независимо от того, какими средствами программирования они созданы, могут использоваться с любыми компиляторами и в любых программах. Посмотрим, как создаются DLL на языке ассемблера фирмы Microsoft.

DLL-библиотека может быть создана как обычный файл ассемблера. В любой библиотеке присутствует код инициализации, который может получать управление в одном из четырех случаев. Нас будет интересовать практическое применение DLL, поэтому рассмотрим минимальный код, необходимый для написания библиотек:

```
.386
.model flat, stdcall
  option casemap :none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
.code

LibMain proc hInstDLL:DWORD, reason:DWORD, unused:DWORD
  mov     EAX, 1
  ret
LibMain Endp

End LibMain
```

Точка входа DLL-библиотеки может иметь любое имя. Представленный фрагмент кода DLL просто возвращает значение 1 (TRUE). Это простейший вариант библиотеки DLL. Программист может написать свои процедуры, используя этот шаблон. Я покажу на примере, как это сделать.

Разработаем DLL, содержащую две процедуры, — сложения двух чисел Sum2 и вычитания Sub2. Исходный текст библиотеки сохраним в файле Sum2.asm. Нашей конечной целью является создание библиотеки Sum2.dll и демонстрация работы процедур Sum2 и Sub2.

В качестве входных параметров процедура Sum2 принимает значения двух целочисленных переменных i1 и i2, а в качестве результата возвращает их

сумму. Процедура Sub2 выполняет вычитание двух целых чисел. Исходный текст библиотеки приведен в листинге 5.25.

Листинг 5.25. Библиотека DLL, содержащая процедуры Sum2 и Sub2

```
;--------------------- SUM2.ASM -----------------------
.386
.model flat, stdcall
  option casemap :none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib

.code
LibMain proc hInstDLL:DWORD, reason:DWORD, unused:DWORD
  mov     EAX, 1
  ret
LibMain Endp

Sum2 proc
  push    EBP
  mov     EBP, ESP
  mov     EAX, [EBP+8]             ; здесь находится i1
  add     EAX, [EBP+12]            ; здесь находится i2
  pop     EBP
  ret     8
Sum2 endp

Sub2 proc
  push   EBP
  mov    EBP, ESP
  mov    EAX, [EBP+8]
  sub    EAX, [EBP+12]
  pop    EBP
  ret    8
Sub2 endp
End LibMain
```

Компиляция и сборка файла DLL обычно выполняется при помощи BAT-файла. Содержимое такого файла может быть, к примеру, таким:

```
@echo off
if exist Sum2.obj del Sum2.obj
if exist Sum2.dll del Sum2.dll
\masm32\bin\ml /c /coff Sum2.asm
\masm32\bin\Link /SUBSYSTEM:WINDOWS /DLL /DEF:Sum2.def Sum2.obj
dir Sum2.*
pause
```

Естественно, что в каждой конкретной конфигурации операторы файла могут быть другими. Запустим такой BAT-файл на выполнение. Если в исходном тексте файла Sum2 нет ошибок, то на выходе получим файл библиотеки с расширением DLL, т. е. Sum2.dll. Обратите внимание, что помимо Sum2.asm требуется файл Sum2.def. В этом файле описаны процедуры, экспортируемые из библиотеки DLL. У нас две процедуры, и файл Sum2.def будет иметь три записи:

```
LIBRARY Sum2
EXPORTS Sum2
        Sub2
```

Компоновщик генерирует два файла — собственно библиотеку динамической компоновки Sum2.dll и библиотеку импорта Sum2.lib. Назначение Sum2.dll понятно — в ней находятся экспортируемые процедуры `Sum2` и `Sub2`. Зачем тогда нужен файл Sum2.lib?

Дело в том, что приложение может использовать DLL одним из двух способов. Первый способ — подключить библиотеку DLL статически, на этапе загрузки приложения. Однако компоновщик не может просто выделить процедуры из библиотеки DLL и поместить их в исполняемый файл приложения. Это связано со сложностью привязок адресов (address fixups) функций, определенных в DLL. Помочь в разрешении этой проблемы может библиотека импорта LIB, в которой содержится необходимая для компоновщика информация, позволяющая корректно "связать" наше приложение с DLL.

Хочется напомнить, что подобный вариант использования DLL мы видим на примерах разработки полнофункциональных приложений на ассемблере, встречающихся в этой книге. В таких программах мы используем функции WIN API из системных библиотек kernel32.dll, user32.dll и gdi32.dll. Для

вызова функций из DLL в ассемблерный модуль включается информация из соответствующих библиотек импорта. Это делается с помощью директив includelib:

```
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\gdi32.lib
includelib \masm32\lib\user32.lib
```

Если мы хотим использовать библиотеку Sum2.dll, то должны будем включить в программу библиотеку импорта Sum2.lib:

```
includelib Sum2.lib
```

Кроме того, необходимо описать прототипы функций `Sum2` и `Sub2` из DLL:

```
Sum2 PROTO
Sub2 PROTO
```

Это все, что необходимо сделать для статического подключения DLL к приложению. Далее рассмотрим пример, демонстрирующий описанную выше методику.

Разработаем программу, которая будет выводить результат суммирования двух целых чисел в окно приложения при нажатии левой кнопки мыши и результат вычитания — при нажатии правой кнопки. Программа будет использовать процедуры `Sum2` и `Sub2` из библиотеки Sum2.dll.

Исходный текст программы (назовем ее LOADSUM2) приведен далее в листинге 5.26.

Листинг 5.26. Программа, использующая библиотеку импорта LIB для вызова функций из DLL

```
;   ---------------- LOADSUM2.ASM -----------------------
.386
.model flat, stdcall
  option casemap :none                          ; различаем регистр символов

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
```

```
  include \masm32\include\gdi32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

 ; Библиотека импорта, сгенерированная компилятором
   includelib Sum2.lib

  ; Прототипы функций, включая функции из DLL
  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
  Sum2 PROTO                          ; функция Sum2 из библиотеки Sum2.dll
  Sub2 PROTO                          ; функция Sub2 из библиотеки Sum2.dll

.data
  szClassName   DB  "LoadDLL_Class", 0
  szDisplayName DB  "ЗАГРУЗКА DLL ПРИ СТАРТЕ ПРИЛОЖЕНИЯ", 0
  CommandLine   DD  0
  hWnd          DD  0
  hInstance     DD  0

  i1            DD  23
  i2            DD -54

  isum          DD  0
  isub          DD  0
   lpFmt         DB  "%s%d", 0
  sInts         DB  "ПЕРВОЕ ЧИСЛО = 23, ВТОРОЕ ЧИСЛО = -54"
  lsInts        EQU $-sInts

  s1            DB  "СУММА  = ", 0
  s2            DB  "РАЗНОСТЬ  = ", 0
  buf           DB  32 dup (0)
  stitle1       DB  "ВЫЧИСЛЕНИЕ СУММЫ ДВУХ ЦЕЛЫХ ЧИСЕЛ", 0
  stitle2       DB  "ВЫЧИСЛЕНИЕ РАЗНОСТИ ДВУХ ЦЕЛЫХ ЧИСЕЛ", 0

.code
```

```
start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX
  invoke  GetCommandLine
  mov     CommandLine, EAX

  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, EAX

WinMain proc hInst      :DWORD,
             hPrevInst  :DWORD,
             CmdLine    :DWORD,
             CmdShow    :DWORD

  LOCAL wc  :WNDCLASSEX
  LOCAL msg :MSG

  ; Заполнение структуры WNDCLASSEX требуемыми параметрами
   mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW or CS_BYTEALIGNWINDOW
  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL
   push    hInst
  pop     wc.hInstance
  mov     wc.hbrBackground, COLOR_WINDOW-1
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName
  invoke  LoadIcon, hInst, 500
  mov     wc.hIcon, EAX
  invoke  LoadCursor, NULL, IDC_ARROW
  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0

  invoke  RegisterClassEx, ADDR wc
  invoke  CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                          ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
                          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL
```

```
  mov     hWnd, EAX
  invoke  ShowWindow, hWnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hWnd

; Цикл обработки сообщений
StartLoop:
  push    0
  push    0
  push    NULL
  lea     EAX, msg
  push    EAX
  call    GetMessage
  cmp     EAX, 0
  je      ExitLoop
  lea     EAX, msg
  push    EAX
  call    TranslateMessage
  lea     EAX, msg
  push    EAX
  call    DispatchMessage
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  ret
WinMain endp
 ; Оконная процедура приложения
 WndProc proc hWin   :DWORD,
              uMsg   :DWORD,
              wParam :DWORD,
              lParam :DWORD
  LOCAL hdc   :HDC
  LOCAL rect  :RECT
  LOCAL ps    :PAINTSTRUCT
   cmp     uMsg, WM_PAINT
  jne     next_1

  lea     EDX, ps
  push    EDX
```

```
  push    hWin
  call    BeginPaint
  mov     hdc, EAX
  invoke  TextOut, hdc, 10, 10, ADDR sInts, lsInts
  lea     EDX, ps
  push    EDX
  push    hWin
  call    EndPaint
  ret
next_1:
                                   push    isum
  push    offset s1
  push    offset lpFmt
  push    offset buf
  call    wsprintf
  add     ESP, 16

  invoke  MessageBox, hWin,ADDR buf, ADDR stitle1, MB_OK
  ret
next_2:
  cmp     uMsg, WM_RBUTTONDOWN
  jne     next_3
  push    i2
  push    i1
  call    Sub2
  mov     isub, EAX
   push    isub
  push    offset s2
  push    offset buf
  call    wsprintf
  add     ESP, 16
   invoke  MessageBox, hWin, ADDR buf, ADDR stitle2, MB_OK
  ret
next_3:
  cmp     uMsg, WM_DESTROY
  jne     next_4
  invoke  PostQuitMessage, NULL
  xor     EAX, EAX
```

```
  ret
      WndProc endp

end start
```

Следует обратить ваше внимание на то, как вызываются процедуры `Sum2` и `Sub2` в обработчиках нажатия кнопок мыши. Параметры передаются через стек в соответствии с директивой `stdcall`. Результат возвращается, как обычно, в регистре `EAX` и помещается либо в переменную `isum` (при сложении), либо в переменную `isub` (при вычитании). Для преобразования полученных значений в текстовые строки используется знакомая нам функция WIN API `wsprintf`. Назначение и смысл остальных операторов и команд, думаю, понятен.

На рис. 5.19 и 5.20 изображены окна работающего приложения.

Рассмотрим второй способ использования DLL — без библиотек импорта. В этом случае DLL загружается и выгружается динамически самим приложением, которое для этого использует функции WIN API `LoadLibrary`, `GetProcAddress` и `FreeLibrary`.

`LoadLibrary` получает дескриптор модуля DLL. Если DLL в памяти не присутствует, Windows загружает ее, после чего инкрементирует счетчик использований библиотеки. Функция имеет следующий синтаксис:

```
HANDLE LoadLibrary(LPCTSTR lpLibraryName);
```

где параметр `lpLibraryName` указывает на строку с завершающим нулем, определяющую имя файла загружаемого модуля

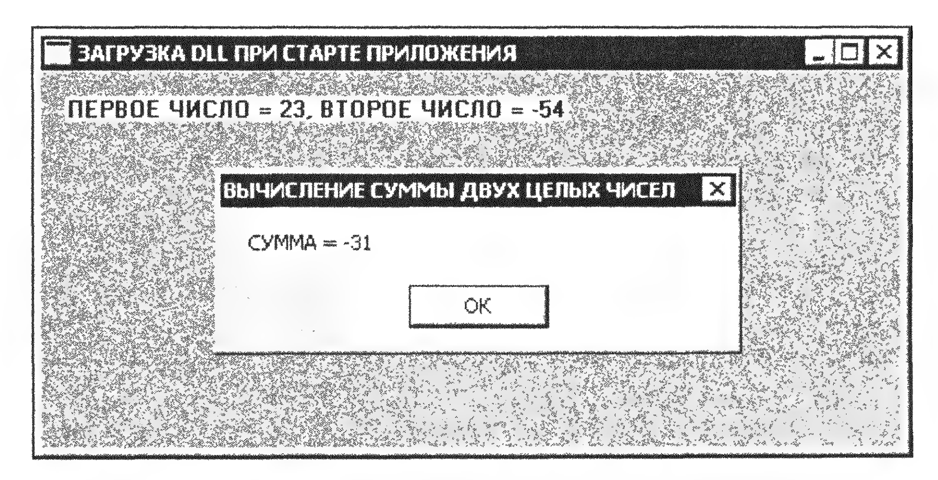


Рис. 5.19. Окно приложения, демонстрирующего вызов функции `Sum2` из DLL при нажатии левой кнопки мыши

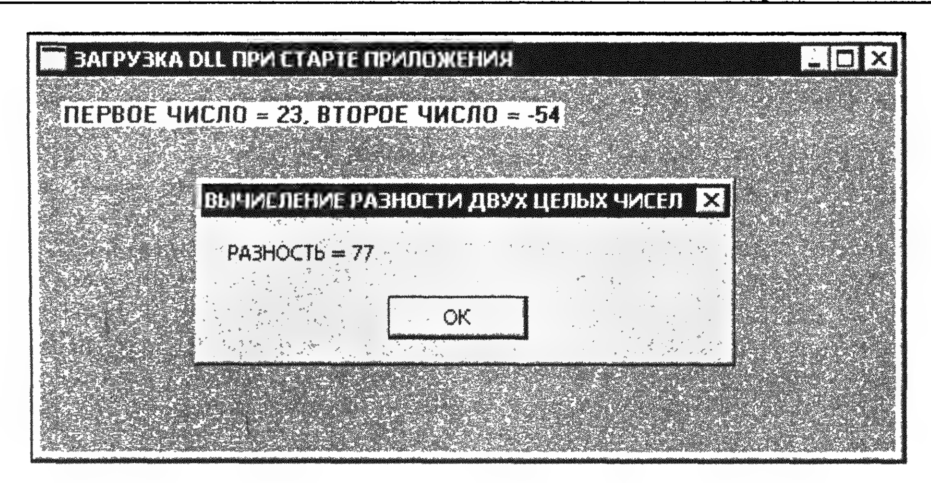


Рис. 5.20. Окно приложения, демонстрирующего вызов функции Sub2 из DLL при нажатии правой кнопки мыши

Функция GetProcAddress применяется для получения адреса функции, находящейся в DLL. Впоследствии приложение может использовать этот адрес для вызова функции. GetProcAddress имеет систаксис:

```
FARPROC GetProcAddress(HMODULE hModule, // дескриптор DLL, возвращенный
                                        // функцией LoadLibrary
                       LPCSTR  lpProcName        // имя функции
                       );
```

При успешном завершении функция возвращает адрес точки входа запрашиваемой процедуры. Наконец, функция FreeLibrary вызывается в том случае, если нужно сообщить Windows, что указанная библиотека DLL приложением более не используется. Windows декрементирует счетчик использований DLL, и когда он становится равным нулю, DLL удаляется из памяти. Функция имеет синтаксис:

```
BOOL FreeLibrary(HMODULE hModule);
```

где hModule — дескриптор модуля DLL.

Последующий пример демонстрирует использование динамической загрузки DLL для вызова процедур Sum2 и Sub2. Исходный текст программы приведен в листинге 5.27.

Листинг 5.27. Программа, демонстрирующая вызов функций из динамически загружаемой DLL

```
; --------------- CALLSUM2.ASM ---------------
.386
.model flat, stdcall
  option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc
  include \masm32\include\gdi32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\gdi32.lib

  ; Прототипы функций

  WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
  WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
  szClassName   DB  "LoadDLL_Class", 0
  szDisplayName DB  "ДИНАМИЧЕСКАЯ ЗАГРУЗКА
sInts         DB  "ПЕРВОЕ ЧИСЛО = 23, ВТОРОЕ ЧИСЛО = -54"
  lsInts        EQU $-sInts

  isum          DD  0
  isub          DD  0
  lpFmt         DB  "%s%d", 0

  s1            DB  "СУММА  = ", 0
  s2            DB  "РАЗНОСТЬ  = ", 0
  buf           DB  32 dup (0)

  stitle1       DB "ВЫЧИСЛЕНИЕ СУММЫ ДВУХ ЦЕЛЫХ ЧИСЕЛ", 0
  stitle2       DB "ВЫЧИСЛЕНИЕ РАЗНОСТИ ДВУХ ЦЕЛЫХ ЧИСЕЛ", 0
```

```
.code
start:
  invoke  GetModuleHandle, NULL
  mov     hInstance, EAX
  invoke  GetCommandLine
  mov     CommandLine, EAX

  invoke  WinMain, hInstance, NULL, CommandLine, SW_SHOWDEFAULT
  invoke  ExitProcess, EAX

WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
  LOCAL wc  :WNDCLASSEX
  LOCAL msg :MSG
  ; Заполнение структуры WNDCLASSEX требуемыми параметрами
  mov     wc.cbSize, sizeof WNDCLASSEX
  mov     wc.style, CS_HREDRAW or CS_VREDRAW or CS_BYTEALIGNWINDOW
  mov     wc.lpfnWndProc, offset WndProc
  mov     wc.cbClsExtra, NULL
  mov     wc.cbWndExtra, NULL
  push    hInst
  pop     wc.hInstance
  mov     wc.hbrBackground, COLOR_WINDOW-1
  mov     wc.hbrBackground, COLOR_WINDOW-1
  mov     wc.lpszMenuName, NULL
  mov     wc.lpszClassName, offset szClassName
  invoke  LoadIcon, hInst, 500
  mov     wc.hIcon, EAX
  invoke  LoadCursor, NULL, IDC_ARROW
  mov     wc.hCursor, EAX
  mov     wc.hIconSm, 0
  invoke  RegisterClassEx, ADDR wc

  invoke CreateWindowEx, WS_EX_OVERLAPPEDWINDOW, ADDR szClassName,\
                         ADDR szDisplayName, WS_OVERLAPPEDWINDOW,\
```

```
                              CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,\
                              CW_USEDEFAULT, NULL, NULL, hInst, NULL
  mov     hWnd,EAX
   invoke  ShowWindow, hWnd, SW_SHOWNORMAL
  invoke  UpdateWindow, hWnd

 StartLoop:
  invoke  GetMessage, ADDR msg, NULL, 0, 0
  cmp     EAX, 0
  je      ExitLoop
  invoke  TranslateMessage, ADDR msg
  invoke  DispatchMessage, ADDR msg
  jmp     StartLoop
ExitLoop:
  mov     EAX, msg.wParam
  WinMain endp

; Оконная процедура

WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
  LOCAL hLib :DWORD
  LOCAL hdc  :HDC
  LOCAL rect :RECT
  LOCAL ps   :PAINTSTRUCT
   cmp     uMsg, WM_PAINT
  jne     next_1

  lea     EDX, ps
  push    EDX
  push    hWin
  call    BeginPaint
  mov     hdc, EAX

  invoke  TextOut, hdc, 10, 10, ADDR sInts, lsInts
  lea     EDX, ps
```

```
  push    EDX
  push    hWin
  call    EndPaint
  ret
next_1:
  cmp     uMsg, WM_LBUTTONDOWN
  jne     next_2
  invoke  LoadLibrary, ADDR libName
  mov     hLib, EAX

  invoke  GetProcAddress, hLib, ADDR FuncSumName
  push    i2
  push    i1
  call    EAX
  mov     isum, EAX
  invoke  FreeLibrary, hLib

  push    isum
  push    offset s1
  push    offset lpFmt
  push    offset buf
  call    wsprintf
  add     ESP, 16
  invoke  MessageBox, hWin, ADDR buf, ADDR stitle1, MB_OK
  ret
next_2:
  cmp     uMsg, WM_RBUTTONDOWN
  jne     next_3
  invoke  LoadLibrary, ADDR libName
  mov     hLib, EAX

  invoke  GetProcAddress, hLib, ADDR FuncSubName
  push    i2
  push    i1
  call    EAX
  mov     isub, EAX
  invoke  FreeLibrary, hLib
  push    isub
  push    offset s2
  push    offset lpFmt
```

```
  push    offset buf
  call    wsprintf
  add     ESP, 16

  invoke  MessageBox, hWin, ADDR buf, ADDR stitle2, MB_OK
  ret
                              WndProc endp

end start
```

Остановлюсь на наиболее интересных фрагментах кода приложения CALLSUM2. Загрузка библиотеки Sum2.dll в память выполняется в следующих строках:

```
invoke  LoadLibrary, ADDR libName
mov     hLib, EAX
```

Возвращаемый функцией LoadLibrary дескриптор модуля запоминается в переменной hLib, которая используется при вызове процедур Sum2 и Sub2:

```
  ...
invoke  GetProcAddress, hLib, ADDR FuncSumName
push    i2
push    i1
call    EAX
mov     isum, EAX
  ...
invoke  GetProcAddress, hLib, ADDR FuncSubName
push    i2
push    i1
call    EAX
mov     isub, EAX
  ...
```

В обоих фрагментах кода функция GetProcAddress возвращает адреса процедур Sum2 и Sub2 в регистре EAX. Поэтому можно использовать команду:

```
call    EAX
```

для вызова процедур по адресу в EAX.

Наконец, рассмотрим консольные приложения Windows.

С консольным типом Windows-приложений мы уже встречались в предыдущих главах. Приложения консоли используются, как правило, тогда, когда для выполнения каких-либо действий не требуется графический интерфейс. Очень часто с помощью консольных приложений выполняют функции перемещения и копирования данных. Многие системные службы спроектированы как консольные приложения.

Мне хотелось бы привести в качестве примера простейшую программу на ассемблере, выполняющую копирование данных из одного файла в другой.

В этом примере используются функции WIN API `CreateFile`, `ReadFile` и `WriteFile` для работы с файлами. В общем случае копирование файлов выполняется в несколько этапов. Вначале открывается исходный файл для считывания данных в буфер памяти. Затем создается файл, куда будут записаны данные. После этого выполняется собственно копирование данных. Наконец, по завершению операции копирования файлы должны быть закрыты.

Рассмотрим синтаксис функций файлового ввода-вывода. Начнем с `CreateFile`. Функция объявляется следующим образом:

```
HANDLE CreateFile(LPCTSTR lpFileName,              // имя файла
                  DWORD   dwAccess,                // тип доступа к файлу
                  DWORD   dwShareMode,             // способ совместного
                                                   // доступа к файлу
  LPSECURITY_ATTRIBUTES   lpSecurityAttributes,    // указатель на структуру
                                                   // SECURITY_ATTRIBUTES
                  DWORD   dwCreationDisposition,   // способ создания файла
                  DWORD   dwFlagsAndAttributes,    // аттрибуты создания
                                                   // файла
                  HANDLE  hTemplateFile           // дескриптор шаблона файла
                  );
```

Функция `CreateFile` в случае успешного завершения возвращает дескриптор вновь созданного объекта. Функция `ReadFile` считывает данные из файла, начиная с позиции, обозначенной указателем файла. После считывания данных указатель файла сдвигается на число считанных байтов. Синтаксис этой функции:

```
BOOL ReadFile(HANDLE  hFile,                 // дескриптор файла
              LPVOID  lpBuffer,              // буфер данных
```

```
            DWORD   nNumberOfBytesToRead,  // количество байт,
                                           // которое необходимо считать
            LPDWORD lpNumberOfBytesRead,   // количество прочитанных байт
            LPOVERLAPPED lpOverlapped      // указатель на
                                           // структуру OVERLAPPED
            );
```

Для записи данных в файл используется функция `WriteFile`. Она имеет синтаксис:

```
BOOL WriteFile(HANDLE  hFile,                    // дескриптор файла
               LPCVOID lpBuffer,                 // буфер данных
               DWORD   nNumberOfBytesToWrite,    // количество байт,
                                             // которые необходимо записать
               LPDWORD lpNumberOfBytesWritten,   // количество фактически
                                                 // записанных байт
               LPOVERLAPPED lpOverlapped     // указатель на
                                             // структуру OVERLAPPED
 );
```

Исходный текст приложения (назовем его `CPFILE`) приведен в листинге 5.28.

Листинг 5.28. Консольное приложение, копирующее данные одного файла в другой

```
;------------------ CPFILE.ASM ----------------------
.386
.model flat, stdcall
  option casemap :none
  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib

.data
  src            DB  "src.txt", 0
  dst            DB  "dst.txt", 0
```

```
  BUF_SIZE      EQU 512
  buf           DB  1024 dup (0)
  BytesRead     DD  0
  BytesWritten  DD  0
  sHandle       DD  0
  dHandle       DD  0

.code

start:
  ; попытка открыть файл с именем в src для чтения

  push    0
  push    FILE_ATTRIBUTE_NORMAL
  push    OPEN_EXISTING
  push    0
  push    0
  push    GENERIC_READ
  push    offset src
  call    CreateFile

  ; если удалось открыть файл, сохраняем полученный дескриптор
  ; в переменной sHandle, иначе выходим из программы

  cmp     EAX, INVALID_HANDLE_VALUE
  je      ex
  mov     sHandle, EAX

  ; попытка создания файла с именем в dst для записи

  push    0
  push    FILE_ATTRIBUTE_NORMAL
  push    CREATE_ALWAYS
  push    0
  push    0
  push    GENERIC_WRITE
  push    offset dst
  call    CreateFile
```

```
  cmp     EAX, INVALID_HANDLE_VALUE

  ; если файл удалось создать, сохраняем его дескриптор
  ; в переменной dHandle, иначе выходим из программы

  je      ex
  mov     dHandle, EAX

; цикл, в котором выполняется копирование файлов
cpy_loop:
  push    0
  push    offset BytesRead
  push    BUF_SIZE
  push    offset buf
  push    sHandle

  ; чтение данных файла-источника в буфер памяти

  call    ReadFile

  ; Если все прочитано, то выходим из цикла
  ; Иначе записываем прочитанные данные в файл-приемник

  cmp     BytesRead, 0
  je      end_cpy

  push    0
  push    offset BytesWritten
  push    BytesRead
  push    offset buf
  push    dHandle
  call    WriteFile
  jmp     cpy_loop
end_cpy:

  ; после копирования закрываем файлы с помощью функции CloseHandle

  push    sHandle
```

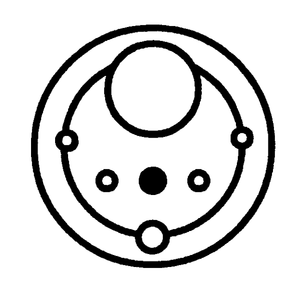
```
  call    CloseHandle
  push    dHandle
  call    CloseHandle
ex:
  push    0
  call    ExitProcess

end start
```

Компиляция приложения может выполняться с использованием тех же опций командной строки, что и для обычных графических приложений.

Заканчивая главу, хотелось бы отметить, что мы рассмотрели далеко не все аспекты применения ассемблера для разработки Windows-приложений. Но даже на этих примерах читатель сможет оценить широкие возможности этого языка программирования.

Глава 6



Встроенный ассемблер языков высокого уровня: принципы использования

Эта глава посвящена применению встроенных средств программирования языков высокого уровня для оптимизации приложений. Насколько важен этот вопрос, можно судить хотя бы по тому факту, что фирмы-изготовители высокоуровневых средств разработки приложений сделали встроенный ассемблер частью среды разработки.

На заре развития аппаратно-программных средств компьютеров наличие языка низкого уровня в составе высокоуровневых языков программирования, таких как C и Pascal, позволяло осуществлять управление ПК с высокой эффективностью. Операционная система MS-DOS давала возможность пользовательским приложениям полностью контролировать персональный компьютер, а сочетание ассемблера и языков высокого уровня в программах позволяло разработчикам писать высокопроизводительные программы.

С приходом операционной системы Windows все изменилось. Программа все еще могла использовать ассемблер для управления и аппаратурой компьютера, и, частично, операционной системой, однако только в Windows 95 / 98 / Me. Другие операционные системы, такие как Windows NT / 2000 / XP, резко ограничили возможности пользователя контролировать работу как операционной системы, так и аппаратуры самого ПК. Казалось, что роль ассемблера в разработке программ, равно как и в повышении эффективности работы приложений, сошла на нет.

Встроенный ассемблер многих языков высокого уровня в середине 90-х воспринимался скорее как дань прошлому, чем как серьезное средство разработки программ. Однако со временем выяснилось, что языки высокого уровня, несмотря на обширные библиотеки функций, в большинстве случаев генерировали не очень эффективный код. Как это ни казалось странным, но новые поколения процессоров требовали новых подходов к проблеме оптимизации, а это мог обеспечить только язык низкого уровня. Кроме того *с появлением операционных систем* Windows NT / 2000 / XP резко обостри-

лась проблема работы приложений в режиме реального времени. Все эти причины заставили ведущие фирмы-производители, такие как Microsoft, Borland, IBM и Intel, усовершенствовать встроенный ассемблер средств разработки на языках высокого уровня.

В настоящее время дискуссии о том, нужен ли ассемблер разработчикам приложений на языках высокого уровня, уже не ведутся, т. к. стало понятно, что этот язык является неотъемлемой частью всех программ и одним из основных средств улучшения производительности приложений на языках высокого уровня.

6.1. Применение встроенного ассемблера Delphi 7

Вначале мы рассмотрим встроенный ассемблер (Built-In Assembler — BASM), используемый в приложениях на Delphi 7. Среда разработки Delphi 7 имеет много общего со своим предшественником Турбо Паскалем.

Синтаксис встроенного ассемблера наследует многие принципы и подходы классических языков, таких как Microsoft MASM или Borland TASM, хотя и имеет определенные отличия. Поскольку встроенный ассемблер является частью среды программирования, то он подчиняется определенным соглашениям, принятым в ней.

Встроенный ассемблер Delphi 7 позволяет:

- использовать большую часть синтаксических конструкций языков Borland TASM и Microsoft MASM. Поддерживаются все команды математического сопроцессора и некоторые выражения Турбо ассемблера;
- использовать все команды процессора Pentium III, включая расширения для работы с большими массивами данных целого и вещественного типа;
- разрабатывать ассемблерные процедуры в теле основной программы;
- использовать идентификаторы переменных, констант и функций Delphi.

Последовательность ассемблерных команд обычно оформляется в виде блока. Команды располагаются между ключевыми словами `asm` и `end`. Схематично блок команд на ассемблере выглядит так:

```
asm
  <команды ассемблера>
end
```

В отличие от программ на классическом ассемблере точка с запятой не является признаком начала комментария во встроенном ассемблере. Ком-

ментарии внутри ассемблерного блока подчиняются правилам, принятым в Delphi.

6.2. Директивы встроенного ассемблера

Встроенный ассемблер поддерживает три директивы: DB (Define Byte — определить байт), DW (Define Word — определить слово) и DD (Define Double Word — определить двойное слово). Каждая из этих директив генерирует данные следующих типов.

1. Директива DB определяет один или последовательность нескольких байт. Операнд должен быть либо числовым значением в диапазоне −128 и 255, либо представлять собой строку символов произвольной длины. Числовое значение генерирует один байт кода, в то время как строка представляет собой последовательность ASCII символов.
2. Директива DW определяет последовательность слов. Каждый операнд должен быть либо числовым значением, лежащим в диапазоне −32 768 и 65 535, либо адресным выражением.
3. Директива DD определяет последовательность двойных слов. Каждый операнд может быть либо числовым значением, лежащим между − 2 147 483 648 и 4 294 967 295, либо адресным выражением.

Далее приведены примеры использования директив DB, DW и DD:

```
asm
  DB    FFH                          // один байт
  DB    0, 99                        // два байта
  DB    'A'                          // символ 'A'
  DB    'Привет, мир !', 0DH, 0AH    // строка символов
  DB    12, "string"                 // Delphi-строка

  DW    0FFFFH                       // одно слово
  DW    0, 9999                      // два слова
  DW    'A'                          // то же, что и DB 'A', 0
  DW    'BA'                         // то же, что и DB 'A', 'B'

  DD    0FFFFFFFFH                   // одно двойное слово
  DD    0, 999999999                 // два двойных слова
  DD    'A'                     // то же, что и DB 'A', 0, 0, 0
  DD    'DCBA'                  // то же, что и DB 'A', 'B', 'C', 'D'
end;
```

Некоторые директивы, такие как EQU, PROC, STRUC, SEGMENT и MACRO, встроенным ассемблером не поддерживаются, хотя для них существуют эквивалентные конструкции в Object Pascal. Например, директиве EQU соответствует константа в языке Pascal, директиве PROC —объявление процедуры или функции, директиве STRUC — записи (record).

Все переменные должны быть определены в соответствии с синтаксисом Delphi. Переменные размером в 1 байт, слово или двойное слово соответствуют идентификаторам BYTE, WORD и Integer. Это продемонстрировано в следующем фрагменте программного кода:

```
var
  varByte: BYTE;
  varWord: WORD;
  varInt:  Integer;
   ...
asm
  mov    AL, varByte
  mov    BX, varWord
  mov    ECX, varInt
end;
```

6.3. Выражения во встроенном ассемблере

Любое выражение встроенного ассемблера характеризуется типом или, более точно, размером. Например, переменная типа Integer имеет размер 4 байта. Встроенный ассемблер всегда выполняет проверку типов переменных. Так при выполнении следующего фрагмента кода:

```
var
  Flag:  Boolean;
  BufW:  WORD;
  BufDw: DWORD;
   ...
asm
  mov    AL, Flag
  mov    BX, BufW
  mov    CX, BufDw
end;
```

компилятор выдаст ошибку при анализе команды:

```
mov     CX, BufDw
```

из-за несовпадения размеров операндов. Возможным выходом в этой ситуации является либо использование 32-разрядного регистра, например ECX, либо явное указание типа второго операнда. Возможные варианты показаны в следующем примере:

```
var
  Flag:  Boolean;
  BufW:  WORD;
  BufDw: DWORD;
asm
  mov     CX, word ptr BufDw
  mov     CX, Word(BufDw)
  mov     CX, BufDw.Word
end
```

При анализе этих трех команд необходимо учитывать то, что в качестве второго операнда используется младшее слово переменной BufDw.

В некоторых случаях, когда в командах используются регистры и ячейки памяти, ассемблер определяет размер переменной в памяти автоматически, в соответствии с типом регистра:

```
procedure TestSize(var Buf);
begin
   ...
  asm
    mov     AL, [Buf]
    mov     CX, [Buf]
    mov     EDX, [Buf]
  end;
    ...
end;
```

Первая команда помещает 1 байт данных в регистр AL из области памяти, определяемой указателем Buf, вторая помещает слово в регистр CX и, на-

конец, третья сохраняет в регистре EDX двойное слово из той же области памяти.

В тех случаях, когда ассемблер не может определить тип данных, необходимо явным образом определить размер операнда, как, например, в этих командах:

```
inc     word ptr [ECX]
mul     word ptr [EDX]
```

Далее дается размер зарезервированных слов встроенного ассемблера (табл. 6.1).

Таблица 6.1. *Зарезервированные слова встроенного ассемблера*

Слово	Размер в байтах
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Правила использования регистров процессора в выражениях встроенного ассемблера такие же, как и для внешних процедур, объявленных с помощью директивы `external`. В начале блока `asm-end` желательно сохранить регистры ESI, EDI, ESP, EBP и EBX. Регистры EAX, ECX и EDX могут использоваться программистом по его усмотрению.

Все выражения встроенного ассемблера приводятся к 32-разрядному целочисленному типу. Не поддерживаются выражения, включающие вещественные значения и строковые переменные, за исключением строковых констант.

Следует отметить отличия между выражениями Object Pascal и встроенного ассемблера, касающиеся способов обработки констант и переменных. Если выполнить следующий фрагмент кода:

```
const
  X = 15;
  Y = 25;
var
  Z: Integer;
```

```
begin
  Z := X + Y;
   ...
```

то ассемблерная команда

```
asm
  mov     Z, X+Y
end;
```

отработает корректно. В то же время, если X и Y — переменные, то эта же команда даст неправильный результат. Дело в том, что встроенный ассемблер не сможет вычислить значение выражения X + Y во время компиляции. Чтобы правильно вычислить сумму, следует использовать следующую последовательность команд:

```
asm
  mov     EAX, X
  add     EAX, Y
  mov     Z, EAX
end;
```

В Object Pascal обращение к переменной означает обращение к ее значению. Обращение к переменной во встроенном ассемблере означает обращение к адресу, где находится переменная. Например, выражение X + 2 в Object Pascal, где X — переменная, означает сумму X и 2. Встроенный ассемблер трактует выражение X + 2 как содержимое ячейки памяти с адресом на 2 байта больше, чем X. Поэтому следующая команда

```
asm
  mov     EAX, X+2
end;
```

вместо суммы переменной X и 2 загрузит в регистр EAX значение ячейки памяти по адресу X + 2. Правильно будет выполняться следующий фрагмент кода:

```
asm
  mov     EAX, X
  add     EAX, 2
end;
```

Следующий пример раскрывает более подробно особенности операций с переменными. Требуется найти сумму первого элемента целочисленного массива X и числа 4. Решим эту задачу двумя способами, причем отобразим результат вычисления в двух разных окнах.

Для этого разместим на главной форме приложения два поля редактирования Edit и две кнопки Button. Напишем программный код обработчиков нажатия кнопок. Исходный текст программы приведен далее в листинге 6.1.

Листинг 6.1. Программа, демонстрирующая особенности работы с переменными встроенного ассемблера

```
unit demopas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1  : TEdit;
    Edit2  : TEdit;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  X: array [1..5] of Integer = (2, -5, 8, 1, -4);
  IX: Integer;

Implementation
```

```
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  asm
    mov     EAX, DWORD PTR X+4
    mov     IX,  EAX
  end;
  Edit1.Text := IntToStr(IX);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  asm
    mov     EAX, DWORD PTR X
    add     EAX, 4
    mov     IX,  EAX
  end;
  Edit2.Text := IntToStr(IX);
end;
end.
```

Окно работающего приложения изображено на рис. 6.1.

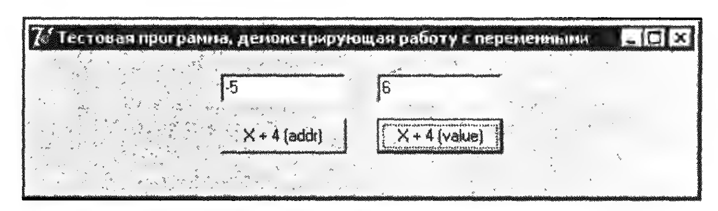


Рис. 6.1. Окно приложения, демонстрирующего принципы работы с переменными и указателями

Внимательно проанализируем код программы. Начнем с обработчика нажатия кнопки `Button1Click`. Результат выполнения кода:

```
mov     EAX, DWORD PTR X+4
mov     IX,  EAX
```

будет отображен в поле редактирования `Edit1` (левое поле на рис. 6.1). Вместо ожидаемого значения 6 мы получили значение второго элемента масси-

ва, т. е. −5. Первая команда ассемблера помещает в регистр EAX значение второго элемента массива. Поскольку массив состоит из целых чисел, то каждый последующий элемент отстоит от предыдущего на 4 байта, следовательно, мы попадаем на второй элемент массива.

Обработчик кнопки Button2 выполняет операцию сложения так, как нам нужно:

```
mov    EAX, DWORD PTR X
add    EAX, 4
mov    IX, EAX;
```

После выполнения этого фрагмента кода в поле редактирования Edit2 будет выведен правильный результат, т. е. 6. Первая команда загружает в регистр EAX значение первого элемента массива, а вторая прибавляет к содержимому этого регистра 4.

6.4. Использование меток во встроенном ассемблере

Метки используются во встроенном ассемблере так же, как и в Object Pascal. Они должны быть объявлены в разделе деклараций блока begin-end, содержащего блок asm-end, с помощью зарезервированного слова label. Следующий фрагмент кода, в котором выполняется сравнение двух вещественных чисел и запоминание меньшего из них в переменной xres, приводится в листинге 6.2.

Листинг 6.2 Фрагмент кода, демонстрирующий использование меток

```
  ...
var
  x1, x2, xres: Single;
  Label x1Lx2, x1Gx2;
begin
  x1 := -98.23;
  x2 := -151.87;
  asm
    finit
    fld     DWORD PTR x1
```

```
    fcomp   DWORD PTR x2
    fstsw
    sahf
    fwait
    jb      x1Lx2
    mov     EAX, x2
    jmp     x1Gx2
  x1Lx2:
    mov     EAX, x1
  x1Gx2:
    mov     DWORD PTR xres, EAX
  end;
   ...
end;
  ...
```

Есть только одно исключение из этого правила, которое касается локальных меток. Локальная метка начинается всегда с символа "@", и предварительно ее декларировать не нужно. Она представляет собой последовательность литер, цифр и знаков подчеркивания. Область видимости локальных меток ограничена блоком asm-end. Предыдущий фрагмент кода, в котором вместо обычных используются локальные метки, приведен в листинге 6.3.

Листинг 6.3. Применение локальных меток в блоке asm-end

```
  ...
var
  x1, x2, xres: Single;
begin
  x1 := -98.23;
  x2 := -151.87;
  asm
    finit
    fld     DWORD PTR x1
    fcomp   DWORD PTR x2
    fstsw
    sahf
    fwait
    jb      @x1Lx2
```

```
    mov     EAX, x2
    jmp     @x1Gx2
  @x1Lx2:
    mov     EAX, x1
  @x1Gx2:
    mov     DWORD PTR xres, EAX
  end;
   ...
end;
  ...
```

6.5. Примеры использования встроенного ассемблера в Delphi-приложениях

Для иллюстрации основных моментов использования встроенного ассемблера разработаем несколько простых приложений. Пусть в нашем первом примере требуется найти сумму двух целых чисел. Вначале разработаем вариант программы, в которой нет ассемблерных команд, а используются только операторы Delphi.

Разместим на главной форме три поля редактирования `Edit` с именами `Edit1`, `Edit2` и `Edit3`, а также кнопку `Button`. Ввод операндов должен выполняться с помощью полей редактирования `Edit1`, `Edit2`, вывод результата по нажатию кнопки `Button1` — в поле `Edit3`.

Исходный текст программы приведен в листинге 6.4.

Листинг 6.4. Программа, выполняющая суммирование чисел с помощью обычных операторов Delphi

```
unit basm1pas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
```

```
    Button1: TButton;
    Edit1  : TEdit;
    Label1 : TLabel;
    Edit2  : TEdit;
    Edit3  : TEdit;
    Label2 : TLabel;
    Label3 : TLabel;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  I1, I2, IRES: Integer;

Implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  I1 := StrToInt(Edit1.Text);
  I2 := StrToInt(Edit2.Text);
  IRES := I1 + I2;
  Edit3.Text := IntToStr(IRES);
end;
end.
```

Обработчик нажатия кнопки `Button1` очень прост и содержит всего три оператора. Смысл переменных `I1`, `I2` и `IRES`, как и операторов обработчика, скорее всего, понятен.

Окно работающего приложения изображено на рис. 6.2.

Рис. 6.2. Окно приложения, выполняющего суммирование двух целых чисел

Изменим предыдущий пример так, чтобы в программе можно было использовать встроенный ассемблер. Вариантов может быть несколько, поэтому рассмотрим их по порядку.

Одним из решений может быть замена в обработчике нажатия кнопки `Button1` оператора суммирования:

```
IRES := I1 + I2
```

на ассемблерный блок команд:

```
asm
  mov     EAX, DWORD PTR I1
  add     EAX, DWORD PTR I2
  mov     DWORD PTR IRES, EAX
end;
```

В этом фрагменте кода целочисленные переменные `I1`, `I2` и `IRES` определены как двойные слова. В остальном текст ассемблерного блока несложен. Сложение двух чисел выполняется в регистре `EAX` с использованием команды `add`, а результат помещается в переменную `IRES`.

Обработчик нажатия кнопки с учетом этих изменений приведен в листинге 6.5.

Листинг 6.5. Суммирование двух целых чисел с использованием ассеблерного блока команд в обработчике нажатия кнопки

```
  ...
procedure TForm1.Button1Click(Sender: TObject);
begin
  asm
    mov     EAX, DWORD PTR I1
```

```
    add     EAX, DWORD PTR I2
    mov     DWORD PTR IRES, EAX
  end;
  Edit3.Text := IntToStr(IRES);
end;
   ...
```

Обратите внимание, что оператор:

```
Edit3.Text := IntToStr(IRES);
```

корректно обрабатывает двойное слово `IRES` как переменную типа `Integer`.

Операцию суммирования можно оформить и в виде процедуры. В "классическом" варианте приложения исходный текст будет выглядеть так, как представлено в листинге 6.6.

Листинг 6.6. Использование процедуры для вычисления суммы двух чисел

```
unit basm1pas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;
    Label1 : TLabel;
    Edit2  : TEdit;
    Edit3  : TEdit;
    Label2 : TLabel;
    Label3 : TLabel;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  I1, I2, IRES: Integer;

Implementation

{$R *.dfm}

procedure AddTwo;
begin
  IRES := I1 + I2;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  I1 := StrToInt(Edit1.Text);
  I2 := StrToInt(Edit2.Text);
  AddTwo;
  Edit3.Text := IntToStr(IRES);
end;
end.
```

В этом варианте программы операция суммирования выделена в отдельную процедуру AddTwo. Процедура не имеет входных параметров и не возвращает значение. Ее довольно легко модифицировать при помощи встроенного ассемблера (листинг 6.7).

Листинг 6.7. Суммирование двух чисел в ассемблерном блоке процедуры

```
procedure AddTwo;
begin
  asm
    mov     EAX, DWORD PTR I1
    add     EAX, DWORD PTR I2
    mov     DWORD PTR IRES, EAX
  end;
end;
```

Как видно из исходного текста, процедура, написанная на встроенном ассемблере, использует переменные программы для вычислений и возврата значения. В этом варианте не используется механизм передачи параметров и не требуется отдельный оператор для возврата результата. При всей простоте реализации таких процедур их применение лучше ограничить. Если в программе присутствует несколько процедур, использующих одни и те же переменные, то будет очень трудно отслеживать изменения таких переменных. Обычно это приводит к ошибкам в работе приложения.

Остановимся теперь более подробно на передаче параметров в процедуры. Входными параметрами могут выступать либо переменные, либо их адреса (указатели), либо их комбинация. Передача параметров как значений не изменяет исходные переменные, поскольку передается копия переменной, которая не сохраняется при выходе из процедуры.

Модифицируем процедуру суммирования так, чтобы в качестве входных параметров она принимала переменные `I1` и `I2`. Исходный текст процедуры и обработчика нажатия кнопки, где эта процедура вызывается, представлен в листинге 6.8.

Листинг 6.8. Процедура суммирования двух чисел, использующая параметры

```
   ...
procedure AddTwo(i1, i2: Integer);
begin
  asm
    mov     EAX, DWORD PTR i1
    add     EAX, DWORD PTR i2
    mov     DWORD PTR IRES, EAX
  end;
end;
   ...
procedure TForm1.Button1Click(Sender: TObject);
begin
  I1 := StrToInt(Edit1.Text);
  I2 := StrToInt(Edit2.Text);
  AddTwo(I1, I2);
  Edit3.Text := IntToStr(IRES);
end;
   ...
```

Как видно из листинга, передача параметров и их обработка в процедуре с использованием встроенного ассемблера никаких радикальных отличий по сравнению с обычными процедурами в Delphi не имеет.

Рассмотрим теперь использование процедуры, возвращающей значение. До сих пор наша процедура помещала результат вычисления в переменную `IRES`. Попробуем теперь обойтись без этой переменной, а результат выполнения процедуры поместить в системную переменную `@Result`, специально предназначенную для этих целей.

Исходный текст процедуры будет выглядеть так, как представлено в листинге 6.9.

Листинг 6.9. Процедура суммирования, возвращающая результат в системной переменной `Result`

```
function AddTwo(i1, i2: Integer): Integer;
begin
  asm
    mov     EAX, DWORD PTR i1
    add     EAX, DWORD PTR i2
    mov     @Result, EAX
  end;
end;
```

Исходный текст всей программы в этом случае также изменится. Приведем фрагменты кода, претерпевшие изменения (листинг 6.10).

Листинг 6.10. Фрагмент кода, отображающий изменения в программе, где используется переменная `Result`

```
   ...
var
  Form1: TForm1;
  I1, I2: Integer;

implementation

{$R *.dfm}

function AddTwo(i1, i2: Integer): Integer;
```

```
begin
  asm
    mov     EAX, DWORD PTR i1
    add     EAX, DWORD PTR i2
    mov     @Result, EAX
  end;
end;
   ...
procedure TForm1.Button1Click(Sender: TObject);
begin
  I1 := StrToInt(Edit1.Text);
  I2 := StrToInt(Edit2.Text);
  Edit3.Text := IntToStr(AddTwo(I1, I2));
end;
end.
```

Поскольку процедура `AddTwo` возвращает целочисленное значение в системной переменной `@Result`, то нет необходимости использовать переменную `IRES` для сохранения результата, и ее можно убрать из секции объявления переменных. Мы можем использовать идентификатор процедуры в качестве аргумента при вызове процедуры `IntToStr` в обработчике нажатия кнопки. Все эти изменения отображены в листинге 6.8.

До сих пор мы работали со значениями переменных. Для обработки массивов данных и строк намного удобней использовать не сами значения, а их адреса в памяти (указатели). Для передачи параметров-указателей используется зарезервированное слово `var`, причем его область действия распространяется в пределах группы переменных до точки с запятой.

Заменим в предыдущем примере целочисленные параметры `i1` и `i2` процедуры `AddTwo` на их адреса. Для этого в начало списка параметров поместим зарезервированное слово `var`. Исходный текст обновленной процедуры приведен в листинге 6.11.

Листинг 6.11. Процедура суммирования двух чисел, принимающая в качестве параметров адреса переменных

```
function AddTwo(var i1, i2: Integer): Integer;
begin
  asm
    mov     EAX, DWORD PTR i1
    mov     EAX, [EAX]
    mov     ECX, DWORD PTR i2
```

```
    add     EAX, [ECX]
    mov     @Result, EAX
  end;
end;
```

Как видим, в ассемблерном блоке добавились некоторые команды. Первая команда:

```
mov     EAX, DWORD PTR i1
```

загружает адрес переменной i1 в регистр EAX. Следующая за ней команда извлекает содержимое по адресу, помещенному в регистр EAX, в этот же регистр.

Для загрузки адреса переменной i2 используется регистр ECX. Команда:

```
add     EAX, [ECX]
```

суммирует содержимое регистра EAX с содержимым ячейки памяти, чей адрес находится в регистре ECX. Код обработчика нажатия кнопки в этом случае не изменится.

Указать процедуре на то, что параметрами являются адреса переменных, можно и другим, часто используемым способом — явно определить параметр как указатель. При этом необходимо указать тип указателя, который соответствует типу переменной.

Например, для переменной типа Integer указателем является PInteger, для переменной вещественного типа указателем является PSingle и т. д. Следовательно, объявление нашей процедуры, если использовать указанные ранее соотношения, будет выглядеть так:

```
function AddTwo(i1, i2: PInteger): Integer;
```

Если используется такое определение параметров процедуры, то код обработчика нажатия кнопки изменится (листинг 6.12).

Листинг 6.12. Фрагмент кода обработчика нажатия кнопки с измененным вариантом вызова процедуры суммирования

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  I1 := StrToInt(Edit1.Text);
  I2 := StrToInt(Edit2.Text);
```

```
  Edit3.Text := IntToStr(AddTwo(@I1, @I2));
end;
```

При вызове процедуры AddTwo следует указывать тип параметра явным образом. Как мы знаем из *главы 3*, для указания адреса переменной необходимо поместить перед ее идентификатором оператор "@" или функцию Addr. В нашем случае используется оператор "@".

Рассмотрим теперь пример, в котором результат выполнения процедуры возвращается в виде адреса переменной. В качестве прототипа будем использовать все ту же процедуру AddTwo. Исходный текст процедуры приведен в листинге 6.13.

Листинг 6.13. Процедура, возвращающая адрес переменной, содержащей сумму двух целых чисел

```
function AddTwo(i1, i2: PInteger): PInteger;
var
  ires: Integer;
begin
  asm
    push   ESI
    mov    EAX, DWORD PTR i1
    lea    ESI, DWORD PTR ires
    mov    EAX, [EAX]
    mov    ECX, DWORD PTR i2
    add    EAX, [ECX]
    mov    DWORD PTR [ESI], EAX
    mov    @Result, ESI
    pop    ESI
  end;
end;
```

Передача параметров в эту процедуру выполняется так же, как и в предыдущем примере. Процесс суммирования понятен, а вот каким образом результат выполнения процедуры возвращается основной программе, имеет смысл рассмотреть более подробно. Для того чтобы вернуть адрес целочисленной переменной, вначале нужно ее объявить. В нашей процедуре такой переменной является ires, объявленная в секции var.

Адрес этой переменной загружается в регистр ESI с помощью команды:

```
lea    ESI, DWORD PTR ires
```

После всех вычислений помещаем результат сложения в ячейку памяти, соответствующую переменной `ires`, и возвращаем адрес самой переменной в основную программу. Эти действия выполняются следующими командами:

```
mov     DWORD PTR [ESI], EAX
mov     @Result, ESI
```

Этот пример очень важен, поэтому в листинге 6.14 приведен полный текст программы.

Листинг 6.14. Программа, использующая для вычисления суммы чисел процедуру с указателями

```
unit basm1pas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;
    Label1 : TLabel;
    Edit2  : TEdit;
    Edit3  : TEdit;
    Label2 : TLabel;
    Label3 : TLabel;

    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
  I1, I2: Integer;

implementation

{$R *.dfm}

function AddTwo(i1, i2: PInteger): PInteger;
var
  ires: Integer;
begin
  asm
    push   ESI
    mov    EAX, DWORD PTR i1
    lea    ESI, DWORD PTR ires
    mov    EAX, [EAX]
    mov    ECX, DWORD PTR i2
    add    EAX, [ECX]
    mov    DWORD PTR [ESI], EAX
    mov    @Result, ESI
    pop    ESI
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  I1 := StrToInt(Edit1.Text);
  I2 := StrToInt(Edit2.Text);
  Edit3.Text := IntToStr(AddTwo(@I1, @I2)^);
end;
end.
```

Как мы используем возвращаемое процедурой значение? Рассмотрим строку обработчика нажатия кнопки:

```
Edit3.Text := IntToStr(AddTwo(@I1, @I2)^);
```

Процедура `AddTwo` принимает в качестве параметров адреса переменных `I1` и `I2`. Возвращаемый процедурой результат, как мы знаем, является указате-

лем на переменную `ires`, объявленную в теле процедуры. Однако процедура `IntToStr` в качестве параметра требует не адреса, а значения целочисленной переменной. Поэтому необходимо выполнить операцию разыменования (dereferencing) указателя переменной `ires`. Для выполнения этой операции необходимо поместить символ "^" после идентификатора процедуры `AddTwo`. В результате такой операции мы получаем значение переменной, находящееся по заранее определенному адресу.

Рассмотренные примеры представляют собой простые программы и демонстрируют технику применения BASM. Следующие примеры намного сложнее и требуют знания дополнительных возможностей встроенного ассемблера.

Поскольку ассемблер прекрасно оптимизирует вычисления в массивах данных, то целесообразно рассмотреть пример, в котором присутствуют операции над такими данными. Пусть требуется найти сумму элементов массива целых чисел и отобразить результат в поле редактирования. Зададим размерность массива равной 7.

Для разработки приложения используем, как и в большинстве задач, главную форму с размещенными на ней полей редактирования `Edit` и кнопкой `Button`. Исходный текст программы представлен в листинге 6.15.

Листинг 6.15. Программа, выполняющая подсчет суммы элементов в массиве целых чисел

```
unit sarpas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;

    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
  IARRAY: array [1..5] of Integer = (45, -23, -5, 11, 7);
  IL: Integer;
  ISUM: Integer;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  IL := SizeOf(IARRAY) div 4;
  asm
    push   ESI
    mov    ESI, offset IARRAY
    mov    ECX, DWORD PTR IL
    dec    ECX
    finit
    fild   DWORD PTR [ESI]
  @L1:
    fiadd  DWORD PTR [ESI+4]
    add    ESI, 4
    loop   @L1
    fistp  DWORD PTR ISUM
    fwait
    pop    ESI
  end;
  Edit1.Text := IntToStr(ISUM);
end;
end.
```

Вычисление суммы элементов выполняется в блоке `asm-end` с использованием команд математического сопроцессора. Доступ к элементам массива осуществляется через регистр `ESI`, в который командой:

```
mov     ESI, offset IARRAY
```

загружается адрес массива, равный адресу его первого элемента. В стек сопроцессора загружается первый элемент массива, после чего в каждой ите-

рации цикла содержимое стека суммируется с последующим элементом массива. Каждый из элементов массива занимает в памяти 4 байта, поэтому следующий операнд адресуется командой:

```
add     ESI, 4
```

По окончанию вычислений содержимое вершины стека запоминается в переменной ISUM командой:

```
fistp   DWORD PTR ISUM
```

одновременно с очисткой вершины стека сопроцессора.

Обратите внимание на использование меток в программе. У нас всего одна метка @L1 цикла, организованного командой loop. Поскольку метка является локальной (используется внутри блока asm-end), то предварительно объявлять ее не нужно.

Попробуем изменить исходный текст ассемблерного блока таким образом, чтобы его можно было применить в отдельной процедуре. Для начала определим, какие входные параметры будут использоваться. Если еще раз проанализировать исходный текст программы, то можно прийти к выводу, что в качестве входных параметров процедуры можно передавать адрес массива и его размер. В качестве результата в основную программу можно вернуть значение суммы элементов. Исходный текст приложения с учетом сделанных изменений выглядит, как показано в листинге 6.16.

Листинг 6.16. Программа подсчета суммы элементов массива с использованием отдельной процедуры

```
unit sarray;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;
```

```
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  iarray: array [1..5] of Integer = (-14, 17, -5, 1, 7);
  IL: Integer;
  isum: Integer;

implementation

{$R *.dfm}

function SumArray(var iaddr; cnt: Integer): PInteger;
begin
  asm
    push   ESI
    mov    ESI, DWORD PTR iaddr
    mov    ECX, DWORD PTR cnt
    dec    ECX
    finit
    fild   DWORD PTR [ESI]
  @L1:
    fiadd  DWORD PTR [ESI+4]
    add    ESI, 4
    loop   @L1
    fistp  DWORD PTR isum
    fwait
    mov    EAX, offset isum
    mov    @Result, EAX
    pop    ESI
  end;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  IL := SizeOf(iarray) div 4;
  Edit1.Text := IntToStr(SumArray(iarray, IL)^);
end;
end.
```

Процедура SumArray теперь содержит блок ассемблерных команд.

Использование отдельно оформленных процедур для однотипных вычислений упрощает структуру программы, однако может замедлить быстродействие. Поэтому оптимальным вариантом улучшения производительности без потери информативности является использование процедур, написанных на языке низкого уровня. Как мы видим на этом примере, математические операции с данными на языке высокого уровня вполне успешно могут решаться при помощи встроенного ассемблера.

6.6. Ассемблерные процедуры в Delphi 7

Встроенный ассемблер Delphi 7 обладает еще одной замечательной возможностью. Процедуры можно писать на "чистом" ассемблере. В этом случае ассемблерные команды помещаются в блок asm-end процедуры. При этом отпадает необходимость в использовании системной переменной @Result для возвращения результата. Ключевое слово assembler, которое используется до сих пор в более ранних версиях Delphi для обозначения ассемблерной процедуры, более можно не использовать. Рассмотрим подробнее некоторые аспекты написания таких процедур.

Возьмем в качестве примера процедуру сложения двух чисел. На ассемблере она могла бы выглядеть так, как представлено в листинге 6.17.

Листинг 6.17. Процедура сложения двух чисел на ассемблере

```
function AddTwo(i1, i2: Integer): Integer;
asm
  mov     EAX, DWORD PTR i1
  add     EAX, DWORD PTR i2
end;
```

Сравните ее с процедурой, использующей блок begin-end (листинг 6.18).

Листинг 6.18. Процедура сложения двух чисел с использованием блока begin-end

```
function AddTwo(i1, i2: Integer): Integer;
begin
  asm
    mov     EAX, DWORD PTR i1
    add     EAX, DWORD PTR i2
    mov     @Result, EAX
  end;
end;
```

Для ассемблерных процедур компилятор выполняет следующие действия по оптимизации:

- при копировании параметров, представляющих собой значения переменных, в локальные переменные никакой дополнительный код не генерируется;
- для возвращения результата выполнения процедуры не используется системная переменная `@Result`, кроме тех случаев, когда возвращаемым значением является строка;
- компилятор генерирует код пролога и эпилога процедуры, если используется фрейм стека. Это выполняется для вложенных процедур и процедур, использующих локальные параметры или стек. В общем виде такая последовательность действий может быть представлена следующим образом:

```
  ...
push    EBP
mov     EBP, ESP
sub     ESP, Locals
  ...
mov     ESP, EBP
pop     EBP
ret     Params
  ...
```

В этом фрагменте идентификаторы `Locals` и `Params` обозначают размеры (в байтах), занимаемые, соответственно, локальными переменными процедуры и параметрами.

Если в процедуру не передаются параметры и не используются локальные переменные, то компилятор генерирует на выходе только команду `ret`. Строки:

```
push    EBP
mov     EBP, ESP
  ...
pop     EBP
```

будут всегда присутствовать, если в процедуру передаются параметры или в самой процедуре задействованы локальные переменные.

Строки:

```
sub     ESP, Locals
  ...
mov     ESP, EBP
```

будут присутствовать в тех случаях, если в процедуре задействованы локальные переменные. Наконец, строка `ret Params` будет присутствовать всегда.

Если в процедуре используются локальные переменные, то при инициализации они всегда устанавливаются в 0.

Ассемблерная процедура возвращает результат в соответствии со следующими правилами:

- порядковые переменные возвращаются в регистре EAX;
- вещественные переменные возвращаются в вершине стека ST(0) математического сопроцессора;
- указатели на переменные (в том числе и на строки) возвращаются в регистре EAX.

Давайте заглянем внутрь ассемблерной процедуры и проанализируем код, который генерирует для нас компилятор. Для этого воспользуемся отладчиком Delphi 7. В качестве объекта анализа возьмем уже знакомую нам процедуру сложения двух целых чисел `AddTwo` (листинг 6.17) и модифицируем исходный листинг так, чтобы можно было складывать вместе 4 числа (листинг 6.19).

Листинг 6.19. Процедура сложения 4-х чисел на ассемблере

```
function AddFour(i1, i2, i3, i4: Integer): Integer;
asm
  mov     EAX, i1
  add     EAX, i2
```

```
  add     EAX, i3
  add     EAX, i4
end;
```

Приложение, использующее процедуру `AddFour`, представляет собой окно Windows с одной кнопкой `Button1` и одним полем редактирования `Edit1`. Исходный текст обработчика нажатия кнопки приведен в листинге 6.20.

Листинг 6.20. Обработчик кнопки `Button1`, в котором выполняется суммирование чисел

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I1, I2, I3, I4, ISum: Integer;
begin
  I1 := 3;
  I2 := -4;
  I3 := 1;
  I4 := 7;
  ISum := AddFour(I1, I2, I3, I4);
  Edit1.Text := IntToStr(ISum);
end;
```

Выполним отладку нашего приложения при помощи встроенного отладчика Delphi 7. Нас будет интересовать тот участок кода, где происходит вызов процедуры `AddFour`. Окно дизассемблера изображено на рис. 6.3.

```
CPU
$0044F144            Thread #$00000D80
   atpas.pas.33: I1 := 3;
   0044F0E8 B803000000        mov eax,$00000003
   atpas.pas.34: I2 := -4;
   0044F0ED BAFCFFFFFF        mov edx,$fffffffc
   atpas.pas.35: I3 := 1;
   0044F0F2 BE01000000        mov esi,$00000001
   atpas.pas.36: I4 := 7;
   0044F0F7 BF07000000        mov edi,$00000007
   atpas.pas.37: ISum := AddFour(I1, I2, I3, I4);
   0044F0FC 57                push edi
   0044F0FD 8BCE              mov ecx,esi
   0044F0FF E840000000        call AddFour
   0044F104 8BF0              mov esi,eax
```

Рис. 6.3. Окно отладчика в точке вызова процедуры `AddFour`

Проанализируем ассемблерные команды, показанные в окне отладчика. Для начала представим дизассемблированный фрагмент кода в более читабельном виде:

```
  ...
mov     EAX, 3
mov     EDX, -4
mov     ESI, 1
mov     EDI, 7
push    EDI
mov     ECX, ESI
call    AddFour
mov     ESI, EAX
  ...
```

Продолжив отладку программы, попадаем внутрь процедуры `AddFour`. Окно отладчика с дизассемблированным кодом изображено на рис. 6.4.

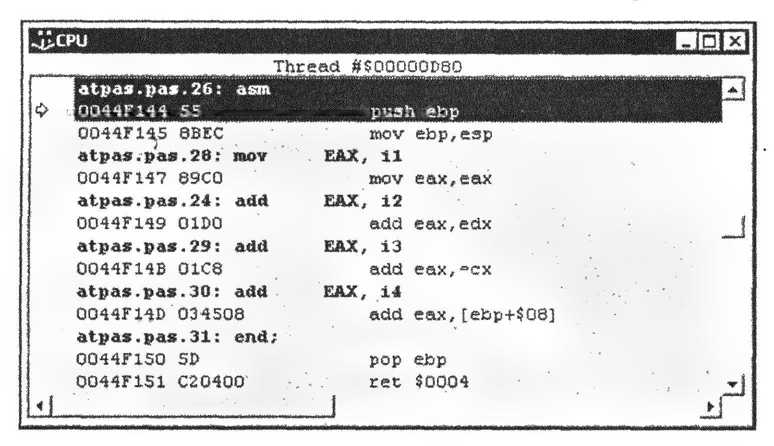


Рис. 6.4. Окно дизассемблированной процедуры `AddFour`

Нас будет интересовать код дизассемблированной процедуры, представленный в листинге 6.21.

Листинг 6.21. Код, сгенерированный отладчиком для процедуры `AddFour`

```
  ...
push    EBP
mov     EBP, ESP
mov     EAX, EAX
add     EAX, EDX
```

```
add     EAX, ECX
add     EAX, [EBP+8]
pop     EBP
ret     4
  ...
```

Сравнивая последние два листинга, можно сделать следующие выводы:

- ассемблерные процедуры в Delphi 7 по умолчанию используют соглашение о передаче параметров типа `register`. Как видно из листинга, первые три параметра передаются в регистрах `EAX(I1)`, `EDX(I2)` и `ECX(I3)`. Четвертый параметр `I4` передается в регистре `EDI` через стек;
- компилятор автоматически генерирует код пролога и эпилога, если для передачи параметров используется стек:

```
push    EBP
mov     EBP, ESP
  ...
pop     EBP
ret     4
```

- процедура возвращает результат в регистре `EAX`.

Модифицируем исходный текст процедуры `AddFour`, точнее, строку с декларацией процедуры следующим образом:

```
function AddFour(i1, i2, i3, i4: Integer): Integer; stdcall;
```

Запустим приложение на отладку. Нас по-прежнему будут интересовать участки кода, где вызывается процедура `AddFour`. В первом окне вы видите фрагмент кода, где выполняется подготовка передачи параметров в процедуру (рис. 6.5).

Интересующий нас фрагмент кода представлен далее в листинге 6.22.

Листинг 6.22. Передача параметров в процедуру `AddFour` с использованием соглашения `stdcall`

```
  ...
mov     EAX, 3
mov     EDX, -4
mov     ESI, 1
mov     EDI, 7
```

```
push    EDI

push    ESI
push    EDX
push    EAX
call    AddFour
mov     ESI, EAX
  ...
```

Как видно из листинга, все параметры передаются через стек, причем последний параметр I4 попадает в стек первым. Возвращаемое процедурой значение помещается в регистр EAX. Внутри самой процедуры параметры обрабатываются так, как показано в окне дизассемблера (рис. 6.6).

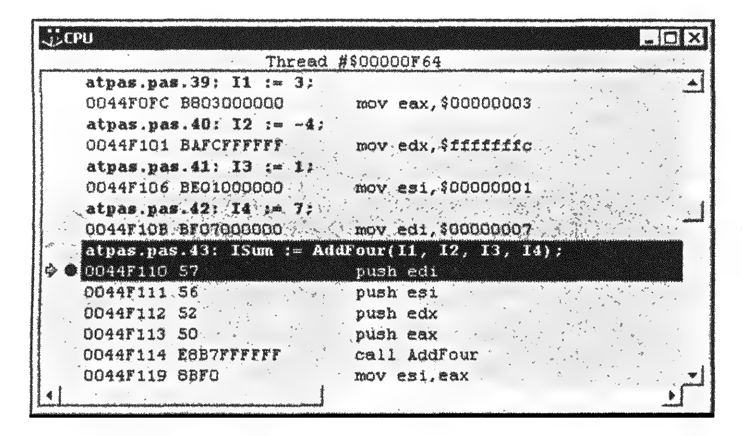


Рис. 6.5. Окно отладчика перед вызовом процедуры AddFour

```
CPU
                    Thread #$00000F64
atpas.pas.26: asm
0044F0D0 55                      push ebp
0044F0D1 8BEC                    mov ebp,esp
atpas.pas.27: mov        EAX, i1
0044F0D3 8B4508                  mov eax,[ebp+$08]
atpas.pas.28: add        EAX, i2
0044F0D6 03450C                  add eax,[ebp+$0c]
atpas.pas.29: add        EAX, i3
0044F0D9 034510                  add eax,[ebp+$10]
atpas.pas.30: add        EAX, i4
0044F0DC 034514                  add eax,[ebp+$14]
atpas.pas.31: end;
0044F0DF 5D                      pop ebp
0044F0E0 C21000                  ret $0010
```

Рис. 6.6. Окно дизассемблера с кодом процедуры AddFour

Представим дизассемблированный текст в более читабельном виде (листинг 6.23).

Листинг 6.23. Обработка данных в процедуре `AddFour` для соглашения `stdcall`

```
  ...
push    EBP
mov     EBP, ESP
mov     EAX, [EBP+8]
add     EAX, [EBP+12]
add     EAX, [EBP+16]
add     EAX, [EBP+20]
pop     EBP
ret     16
  ...
```

Как и следовало ожидать (и это видно из последних двух листингов), передача параметров выполняется в соответствии с директивой `stdcall`, причем компилятор генерирует весьма эффективный код! На основании дизассемблированных листингов можно сделать такой вывод: процедура, написанная полностью на встроенном ассемблере Delphi, практически эквивалентна отдельно написанному ассемблерному модулю. Теперь можно преобразовать процедуру вычисления суммы элементов целочисленного массива `SumArray` "смешанного" типа в полностью ассемблерный вариант. Она будет выглядеть так, как представлено в листинге 6.24.

Листинг 6.24. Ассемблерный вариант процедуры `SumArray` "смешанного" типа

```
function SumArray(var iaddr; cnt: Integer): PInteger;
asm
  push    ESI
  mov     ESI, DWORD PTR iaddr
  mov     ECX, DWORD PTR cnt
  dec     ECX
  finit
  fild    DWORD PTR [ESI]
@L1:
  fiadd   DWORD PTR [ESI+4]
  add     ESI, 4
  loop    @L1
```

```
  fistp   DWORD PTR isum
  fwait
  mov     EAX, offset isum
  pop     ESI
end;
```

Рассмотрим более сложный пример ассемблерной процедуры. Пусть требуется найти сумму элементов массива, начиная с *i*-го элемента и заканчивая *k*-м. Массив состоит из 7 вещественных чисел.

На главной форме приложения должны быть размещены поля редактирования Edit с именем Edit1 для вывода результата и кнопка Button. Вычисление суммы элементов выполняется в ассемблерной процедуре, возвращающей в качестве результата указатель на ячейку памяти, содержащую искомое значение. Обратите внимание, как передаются и используются параметры при вызове процедуры. Исходный текст программы приведен в листинге 6.25.

Листинг 6.25. Программа, выполняющая подсчет суммы элементов в определенном диапазоне изменения индекса массива

```
unit partsum;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;
    Edit2  : TEdit;
    Edit3  : TEdit;
    Label1 : TLabel;
    Label2 : TLabel;
    Label3 : TLabel;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  farray: array [1..7] of Single = (4.3, -1.2, 0.5, 7.45, -6.15,
                                     12.3, -3.85);
  fr, lr: Integer;
  fsum: Single;

implementation

{$R *.dfm}

function SumReals(var fa; fr, lr: Integer): PSingle;
asm
  push    ESI
  mov     ESI, DWORD PTR fa
  mov     EDX, DWORD PTR fr
  mov     ECX, DWORD PTR lr

  sub     ECX, EDX
  shl     EDX, 2
  add     ESI, EDX
  finit
  fld     DWORD PTR [ESI]
@L1:
  fadd    DWORD PTR [ESI+4]
  add     ESI, 4
  loop    @L1
  fstp    DWORD PTR fsum
  fwait
  mov     EAX, offset fsum
  pop     ESI
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  fr := StrToInt(Edit1.Text);
  lr := StrToInt(Edit2.Text);
  Edit3.Text := FloatToStrF((SumReals(farray, fr, lr)^),
                                   ffGeneral, 5, 7);
end;
end.
```

Анализ работы приложения начнем с ассемблерной процедуры SumReals. В качестве входных параметров мы передаем процедуре адрес массива вещественных чисел, порядковый номер первого элемента выделенного диапазона (параметр fr, минимальное значение равно 0, что соответствует началу массива), порядковый номер последнего элемента выделенного диапазона (параметр lr). Адрес массива помещаем в регистр ESI, а порядковые номера первого и последнего элементов, соответственно, в регистры EDX и ECX. Далее вычисляем общее количество суммируемых элементов:

```
sub     ECX, EDX
shl     EDX, 2
add     ESI, EDX
```

После выполнения этих команд в регистре ECX будет содержаться количество суммируемых элементов, а в регистре ESI — смещение первого элемента последовательности. Команда:

```
shl     EDX, 2
```

переводит порядковый номер первого элемента, участвующего в суммировании, в эквивалентное ему смещение в байтах от начала массива, поэтому смысл следующей команды:

```
add     ESI, EDX
```

становится очевиден.

Результирующее значение процедура возвращает в регистре EAX.

В обработчике нажатия кнопки обратим внимание на следующий оператор:

```
Edit3.Text := FloatToStrF((SumReals(farray, fr, lr)^), ffGeneral, 5, 7);
```

Для вывода вещественного числа в поле редактирования `Edit` воспользуемся процедурой `FloatToStrF`. Она встречалась нам в *главе 3*, но вспомним, что эта процедура выполняет преобразование вещественного числа в строку символов с использованием опций форматирования, задаваемых программистом.

Поскольку процедура `SumReals` возвращает адрес переменной, то, выполняя разыменование указателя, получим искомое значение суммы.

Окно работающего приложения изображено на рис. 6.7.

Рис 6.7. Окно приложения, вычисляющего частичную сумму элементов массива вещественных чисел

Можно упростить процедуру вычисления суммы. Встроенный ассемблер позволяет нам получить результат математической операции из вершины стека сопроцессора, минуя регистр `EAX`! После некоторых изменений исходный текст процедуры `SumReals` будет выглядеть так, как представлено в листинге 6.26.

Листинг 6.26. Процедура, возвращающая значение через вершину стека сопроцессора

```
function SumReals(var fa; fr, lr: Integer): Single;
asm
  push    ESI
  mov     ESI, DWORD PTR fa
  mov     EDX, DWORD PTR fr
  mov     ECX, DWORD PTR lr
  sub     ECX, EDX
  shl     EDX, 2
  add     ESI, EDX
  finit
  fld     DWORD PTR [ESI]
```

```
@L1:
  fadd    DWORD PTR [ESI+4]
  add     ESI, 4
  loop    @L1
  fwait
  pop     ESI
end;
```

Обратите внимание на изменения в исходном тексте программы. Во-первых, исчезли следующие команды:

```
fstp    DWORD PTR fsum
mov     EAX, offset fsum
```

Первая команда нужна была для того, чтобы сохранить значение суммы в памяти, вторая — чтобы вернуть адрес переменной, в которой хранится результат, в основную программу. Поскольку в вершине стека сопроцессора находится теперь значение, а не адрес, то нужно заменить тип возвращаемого процедурой результата на `Single`. Заголовок нашей процедуры будет теперь выглядеть так:

```
function SumReals(var fa; fr, lr: Integer): Single;
```

а сама процедура возвращает значение, а не адрес.

Результат вычислений выводится в окно приложения при помощи оператора:

```
Edit3.Text := FloatToStrF((SumReals(farray, fr, lr)), ffGeneral, 5, 7);
```

В следующем примере определим порядковый номер элемента в массиве вещественных чисел. Для разработки этого примера ограничимся массивом из 7 элементов. Наше приложение будет иметь главную форму с размещенными на ней компонентами. Будем использовать два поля редактирования `Edit` с именами `Edit1` и `Edit2`, две метки `Label` и кнопку `Button` с именем `Button1`.

Работающее приложение ожидает ввода вещественного числа в поле редактирования `Edit1`, после чего выполняет поиск элемента в массиве. Если элемент найден, то в поле `Edit2` выводится номер его позиции в массиве. Если элемент не найден, то выводится сообщение `"Value not found!"`. Исходный текст программы представлен в листинге 6.27.

Листинг 6.27. Программа, определяющая позицию элемента в массиве вещественных чисел

```
unit pospas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;
    Edit2  : TEdit;
    Label1 : TLabel;
    Label2 : TLabel;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  farray: array [1..8] of Single = (2.1, 11.9, 3.2, -4.3, 5.4,
                                    6.5, -7.6, -8.7);
  larray: Integer;
  fVal: Single;
  fpos: Integer;

implementation

{$R *.dfm}

function RetPos(var farray; larray: Integer; fVal: Single): Integer;
asm
```

```
  push    ESI
  mov     ESI, DWORD PTR farray
  mov     EDX, ESI
  mov     ECX, DWORD PTR larray

  dec     ECX
  finit
  fldz
  fld     DWORD PTR fVal
@next:
  fcom    DWORD PTR [ESI]
  fstsw   AX
  sahf
  je      @found

  add     ESI, 4
  loop    @next
  xor     EAX, EAX
  jmp     @exit
@found:
  sub     ESI, EDX
  shr     ESI, 2
  mov     EAX, ESI
@exit:
  pop     ESI
  fwait
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  larray:= SizeOf(farray) div 4;
  fVal:= StrToFloat(Edit1.Text);
  fpos:= RetPos(farray, larray, fVal);
  if fpos < larray then
    Edit2.Text:= IntToStr(fpos)
  else
    Edit2.Text:= 'Value not found!';
  end;
end.
```

Проанализируем работу процедуры RetPos. Вначале полученные из основной программы параметры загружаются в соответствующие регистры. Адрес массива загружается в регистры ESI и EDX, размер массива — в ECX. Операция сравнения введенного числа с элементами массива выполняется с использованием математического сопроцессора.

Для организации последовательного перебора элементов массива выполняется цикл loop со значением счетчика в регистре ECX, равным уменьшенному на 1 размеру массива:

```
  ...
@next:
  fcom     DWORD PTR [ESI]
  fstsw    AX
  sahf
  je       @found
  add      ESI, 4
  loop     @next
  xor      EAX, EAX
  jmp      @exit
  ...
```

Исходное вещественное число находится в вершине стека сопроцессора, куда оно загружается командой:

```
fld     DWORD PTR fVal
```

Если число обнаружено в массиве, выполняется переход на метку @found, где вычисляется позиция найденного элемента в массиве:

```
  ...
@found:
  sub      ESI, EDX
  shr      ESI, 2
  mov      EAX, ESI
  ...
```

В регистре EDX находится смещение первого элемента массива, т. е. адрес массива, а в регистре ESI — адрес обнаруженного элемента. Разность содержимого двух этих регистров дает нам смещение (позицию) элемента

в байтах. Поскольку вещественные числа типа `Single` отстоят друг от друга на 4 байта, то сохраненную в регистре `ESI` величину смещения необходимо разделить на 4, что и выполняет команда:

```
shr    ESI, 2
```

Результат возвращается, как обычно, в регистре `EAX` и отображается в поле редактирования.

Обработчик нажатия кнопки осуществляет ввод-вывод информации и интерфейс с ассемблерной процедурой.

Окно работающего приложения изображено на рис. 6.8.

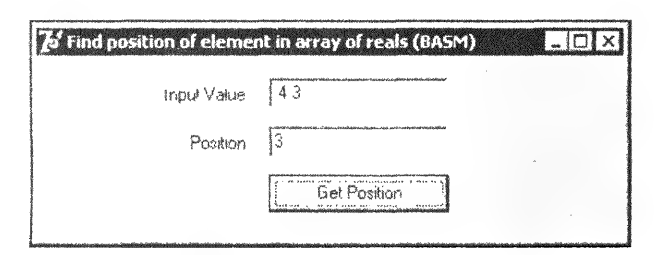


Рис. 6.8. Окно приложения, отображающего позицию элемента в массиве

Следующий пример является, пожалуй, более сложным по сравнению с предыдущими. Имеется два массива вещественных чисел одинаковой размерности. Требуется найти в них несовпадающие элементы.

В результате выполнения приложения на экран будет выведено содержимое массива целых чисел, которые являются номерами позиций несовпадающих элементов в обоих массивах. На главной форме приложения разместим 4 поля редактирования `Edit`, две кнопки `Button` и две метки `Label`.

Исходный текст программы представлен в листинге 6.28.

Листинг 6.28. Программа, выполняющая поиск несовпадающих элементов в двух массивах вещественных чисел

```
unit uneqpas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,  Dialogs, StdCtrls;
```

```
type
  TForm1 = class(TForm)
    Edit1  : TEdit;
    Edit2  : TEdit;
    Button1: TButton;
    Button2: TButton;
    Edit3  : TEdit;
    Edit4  : TEdit;
    Label1 : TLabel;
    Label2 : TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  x1: array [1..7] of Single = (-5.6, -2.3, 23, 6.9, 7, 23, -1);
  x2: array [1..7] of Single = (5.6, -2.3, 22, 9, 7, 3, -1);
  itmp: array [1..7] of Integer;
  xlen, cnt: Integer;

implementation

{$R *.dfm}

procedure FindDif(var x1, x2, itmp; xlen: Integer);
asm
  push    EBX
  push    ESI
  push    EDI
  mov     ESI, x1
  mov     EDI, x2
```

```
  mov     EDX, ESI
  mov     EBX, itmp
  mov     ECX, DWORD PTR xlen
  sub     ESI, 4
  sub     EDI, 4

@again:
  add     ESI, 4
  add     EDI, 4
  mov     EAX, [ESI]
  cmp     EAX, [EDI]
  jne     @store
@next:
  dec     ECX
  cmp     ECX, 0
  jnz     @again
  jmp     @exit

@store:
  mov     EAX, ESI
  sub     EAX, EDX
  shr     EAX, 2
  add     EAX, 1
  mov     [EBX], EAX
  add     EBX, 4
  jmp     @next
@exit:
  pop     EDI
  pop     ESI
  pop     EBX
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  for cnt := 1 to xlen do
    itmp[cnt] := 0;
  for cnt := 1 to xlen do
```

```
  begin
    if x1[cnt] <> x2[cnt] then
      itmp[cnt] := cnt
    else continue;
  end;
  Edit1.Text := ' ';
  for cnt:= 1 to xlen do
  begin
    if itmp[cnt] <> 0 then
      Edit1.Text := Edit1.Text + IntToStr(itmp[cnt]) + ', ';
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  for cnt := 1 to xlen do
    itmp[cnt] := 0;
  FindDif(x1, x2, itmp, xlen);
  Edit2.Text := ' ';
  for cnt := 1 to xlen do
  begin
    if itmp[cnt] <> 0 then
      Edit2.Text := Edit2.Text + IntToStr(itmp[cnt]) + ', ';
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  xlen:= SizeOf(x1) div 4;
  Edit3.Text:= ' ';
  Edit4.Text:= ' ';
  for cnt:= 1 to xlen do
    Edit3.Text := Edit3.Text+FloatToStrF(x1[cnt], ffGeneral, 5, 7)+'  ';
  for cnt:= 1 to xlen do
    Edit4.Text := Edit4.Text+FloatToStrF(x2[cnt], ffGeneral, 5, 7)+'  ';
end;
end.
```

Программа разработана таким образом, чтобы можно было наблюдать результат сравнения двух массивов, используя два независимых обработчика нажатия кнопок. При нажатии на первую кнопку выполняется обработчик, реализованный операторами на языке высокого уровня без использования встроенного ассемблера. При нажатии на вторую кнопку выполняется обработчик, использующий ассемблерную процедуру сравнения.

Окно работающего приложения изображено на рис. 6.9.

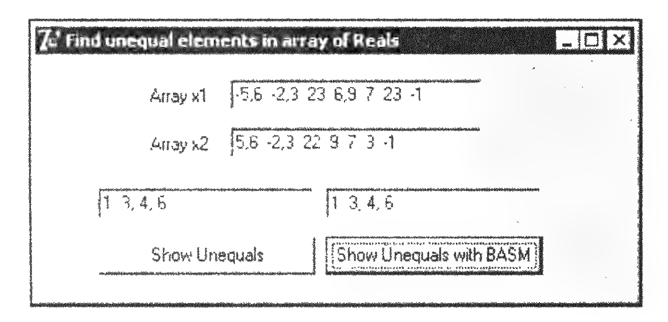


Рис. 6.9. Окно приложения, отображающего несовпадающие элементы двух массивов с помощью двух разных процедур

В нашей программе используется процедура `FindDif`. В качестве параметров она принимает адреса трех массивов: двух сравниваемых массивов `x1` и `x2` и массива целых чисел `itmp`, в который записываются номера позиций в массивах, где значения элементов не совпадают. Еще один параметр, необходимый для работы процедуры — размер массива `xlen`. Поскольку в нашем примере размерность всех массивов одинакова, то не имеет значения, размер какого массива передается в процедуру.

В начале процедуры, как обычно, сохраняем те регистры процессора, которые используются операционной системой. Для работы с элементами массивов `x1` и `x2` помещаем их адреса в регистры `ESI` и `EDI`, сохраняем начальное смещение массива `x1` (оно нам пригодится) в регистре `EDX`. Также нам потребуется и размер `xlen` массива `x1` для организации сравнения элементов в цикле. Его мы сохраняем в регистре `ECX`. Кроме того, нам необходимо знать адрес массива `itmp`. Его мы сохраняем в регистре `EBX`. Все эти действия выполняет следующий фрагмент программного кода ассемблерной процедуры:

```
push    EBX
push    ESI
push    EDI
mov     ESI, x1
```

```
mov     EDI, x2
mov     EDX, ESI
mov     EBX, itmp
mov     ECX, DWORD PTR xlen
```

Для выполнения сравнения элемент массива `x1` помещаем в регистр `EAX`, а элемент массива `x2` находится в памяти, адрес которой содержит регистр `EDI`:

```
mov     EAX, [ESI]
cmp     EAX, [EDI]
jne     @store
```

Если элементы равны, то к адресам текущих элементов добавляется 4, и выполняется очередной цикл сравнения. Если элементы не равны, то выполняется переход на метку `@store`, где вычисляется позиция неравных элементов в массиве, и номер этой позиции загружается в массив `itmp`. Каждый раз после очередной записи он увеличивается на 4, указывая место, куда, возможно, будет загружен следующий элемент. После записи позиции элемента в массив `itmp` и продвижения указателя к следующему адресу цикл сравнения элементов в массивах `x1` и `x2` повторяется. Все эти действия отображены в следующем фрагменте кода:

```
   ...
@store:
  mov     EAX, ESI
  sub     EAX, EDX
  shr     EAX, 2
  add     EAX, 1
  mov     [EBX], EAX
  add     EBX, 4
  jmp     @next
   ...
```

Обычные арифметические команды сравнения чисел в процедуре `FindDif` можно заменить командами математического сопроцессора. В этом случае исходный текст процедуры изменится и будет выглядеть так, как представлено в листинге 6.29.

Листинг 6.29. Процедура поиска несовпадающих элементов, в которой используются команды сопроцессора

```
procedure FindDif(var x1, x2, itmp; xlen: Integer);
asm
  push    EBX
  push    ESI
  push    EDI
  mov     ESI, x1
  mov     EDI, x2
  mov     EDX, ESI
  mov     EBX, itmp
  mov     ECX, DWORD PTR xlen
  sub     ESI, 4
  sub     EDI, 4

  finit
  fldz
@again:
  add     ESI, 4
  add     EDI, 4
  fld     [ESI]
  fcomp   [EDI]
  fstsw   AX
  sahf
  jne     @store
@next:
  dec     ECX
  cmp     ECX, 0
  jnz     @again
  jmp     @exit
@store:
  mov     EAX, ESI
  sub     EAX, EDX
  shr     EAX, 2
  add     EAX, 1
  mov     [EBX], EAX
  add     EBX, 4
  jmp     @next
```

```
@exit:
  fwait
  pop     EDI
  pop     ESI
  pop     EBX
end;
```

Здесь блок команд сравнения элементов выглядит несколько иначе, т. к. используется математический сопроцессор:

```
  ...
  fld     [ESI]
  fcomp   [EDI]
  fstsw   AX
  sahf
  jne     @store
@next:
  ...
```

Команда `fld` загружает число из массива `x1` (его адрес находится в регистре `ESI`) в вершину стека сопроцессора. Следующая команда `fcomp` выполняет сравнение числа в стеке с элементом массива `x2` (его адрес находится в `EDI`). В результате сравнения устанавливаются соответствующие биты (`C3`, `C2`, `C0`) в регистре состояния сопроцессора. Команда `fstsw` сохраняет слово состояния в регистре `AX`. Для удобства манипуляций с битами слова состояния запишем содержимое старшего байт регистра `AX` в регистр флагов при помощи команды `sahf`.

Нам нужно определить равенство или неравенство двух чисел. Для этого достаточно проанализировать бит установки 0 в регистре флагов (`ZF` — zero flag). Если `ZF = 0`, то числа не равны, и наоборот, если `ZF = 1` указывает на равенство чисел. Далее все происходит так, как и в предыдущем варианте этой процедуры.

Хотелось бы обратить ваше внимание на очень важный момент работы этого приложения. Процедура `FindDif` не возвращает никакого результата, тем не менее в массиве `itmp` оказываются нужные значения. Процедуры встроенного ассемблера могут вообще не возвращать никакого значения, а использовать в качестве "разделяемой памяти" переменные и массивы, адреса которых передаются им в качестве параметров. Термин "разделяемая память" взят в кавычки по той причине, что в операционных системах Windows уже существует такое понятие и оно отличается от нашего

определения. Тем не менее этот термин достаточно точно отображает суть дела. Программисты довольно часто используют такую методику для обработки переменных и массивов несколькими процедурами.

6.7. Обработка строк во встроенном ассемблере

Среда программирования Delphi 7 имеет весьма широкие возможности для манипулирования и обработки символов и строк. Delphi поддерживает следующие четыре типа строк:

- тип `ShortString` или короткая строка;
- тип `AnsiString` или длинная строка;
- тип `WideString` или большая строка;
- тип `PChar` или строка с завершающим нулем.

Для объявления коротких и длинных строк можно также использовать зарезервированное слово `String`. Например, объявление:

```
var S: String;
```

определяет строку `S`, тип которой определяется директивой компилятора `$H`. Если используется директива компилятора `$H-`, строка интерпретируется как короткая `ShortString`. Директива компилятора `$H+` интерпретирует строку как `AnsiString`, причем после зарезервированного слова `String` не должно быть квадратных скобок. По умолчанию установлено значение `$H+`, и в наших примерах будем считать, что принята именно эта директива.

Строковые переменные разных типов объявляются следующим образом:

```
var
  ShString:   String[100];     // короткая строка длиной до 100 символов
  ShMaxString: ShortString;    // короткая строка длиной до 255 символов;
  StdString: String;           // длинная строка;
  WdString:   WideString;      // большая строка;
  PcString:   PChar;           // указатель на строку с завершающим нулем;
  ArrString: array [0..100] of Char;   // строка с завершающим нулем
                                       // и длиной до 100 символов
```

Рассмотрим подробно каждый из этих типов строк, а также возможности обработки строк средствами встроенного ассемблера. Начнем с коротких строк.

Короткие строки представляют собой последовательность ASCII символов, и их размер не должен превышать 255 символов. Первый символ последовательности содержит размер строки. Небольшая программа на встроенном ассемблере показывает, как определить размер короткой строки (листинг 6.30).

Листинг 6.30. Программа, определяющая размер короткой строки

```
unit sspas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;
    Edit2  : TEdit;
    Label1 : TLabel;
    Label2 : TLabel;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  s1: ShortString;                      // s1 — короткая строка
  ls1: Integer;

implementation

{$R *.dfm}
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  s1 := Edit1.Text;
  asm
    mov     ESI, offset s1
    mov     AL, BYTE PTR [ESI]
    mov     BYTE PTR ls1, AL
  end;
  Edit2.Text := IntToStr(ls1);
end;
end.
```

Окно работающего приложения изображено на рис. 6.10.

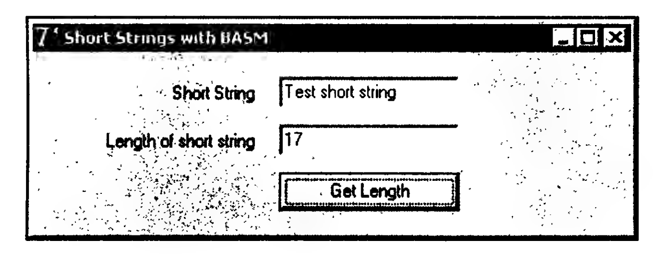


Рис. 6.10. Окно приложения, отображающего размер короткой строки

Короткие строки используются преимущественно в 16-разрядных приложениях и поддерживаются в Delphi 7 только для обратной совместимости (backward compatibility).

Вторым типом строк, который мы рассмотрим, являются длинные строки, или `AnsiString`. Выделение памяти под эти строки происходит динамически, а их размер практически неограничен. Длинная строка описывается 32-разрядным указателем (переменной строки). Delphi автоматически выделяет память для такой строки, а также добавляет в конец строки нулевой символ (для совместимости с WIN API). Поскольку переменная длинной строки представляет собой указатель, то несколько переменных могут ссылаться на одно и то же значение без использования дополнительной памяти. Каждая строка `AnsiString` имеет счетчик ссылок (reference count), в котором содержится количество строковых переменных, ссылающихся на один и тот же адрес. Если переменная типа `AnsiString` модифицирует строку, то счетчик ссылок декрементируется. Когда счетчик ссылок становится равным нулю, блок памяти, занимаемый строкой, освобождается, а указатель строки принимает нулевое значение. Следующий фрагмент кода иллюстрирует все сказанное выше (листинг 6.31).

Листинг 6.31. Фрагмент кода, демонстрирующий прицилы работы с длинными строками

```
  ...
var
  Src, Dst: String; // строки проинициализированы, но память под них
                    // не выделена. Счетчики ссылок для строк равны 0
begin
  Src := 'SOURCE STRING';           // строка размещена в памяти,
                                    // счетчик ссылок равен 1
  Dst := Src;                       // строка Dst ссылается на Src,
                                    //счетчик ссылок для Src равен 2
  Dst := Dst + ' + DEST STRING';    //строка Dst изменилась и скопирована
                                    //в другую область памяти. Значение
                                    //счетчика Src уменьшилось на 1
  ...
```

Большие строки `WideString` представляют собой последовательность 16-битовых символов UNICODE и во многом похожи на длинные строки `AnsiString`. Наиболее существенным недостатком таких строк является отсутствие счетчика ссылок. Это приводит к тому, что при любом присвоении одной строки другой необходимо выделять память и копировать строку в эту область памяти, что приводит к снижению производительности.

Строки `WideString` легко преобразуются в `AnsiString`, и наоборот. Эти действия выполняются компилятором при присвоении. Следующий фрагмент кода (листинг 6.32) показывает, как это делается.

Листинг 6.32. Взаимное преобразование строк `WideString` и `AnsiString`

```
  ...
var
  aString: String;
  wString: WideString;
begin
 wString := 'WideString to AnsiString conversion demo';
 aString := wString;        // преобразование WideString в AnsiString
 aString := 'AnsiString to WideString conversion demo';
 wString := aString;        //преобразование AnsiString в WideString
  ...
```

Мы не будем рассматривать здесь строки WideString отдельно, поскольку в большинстве случаев манипуляции с ними аналогичны тем, которые выполняются для AnsiString. Рассмотрение применения больших строк для операций с многобайтовыми символами и последовательностями выходит за рамки тематики этой книги.

Еще один тип строк, который мы рассмотрим, — строки с завершающим нулем (null-terminated string). Эти строки представляют собой массивы символов, в которых первый символ имеет нулевое смещение. Такая строка не хранит размер, зато имеет завершающий символ 0. Такое представление строк является типичным для языка С. Строки с завершающим нулем широко используются при вызове функций WIN API (последние написаны на С).

Следует сказать, что длинные строки Delphi также имеют завершающий ноль, что делает их совместимыми со строками с завершающим нулем. Длинные строки можно преобразовать к строкам с завершающим нулем, приведя их к типу PChar. В наших последующих примерах мы будем использовать такую возможность.

Встроенный ассемблер позволяет выполнять практически любые сколь угодно сложные манипуляции со строками. Особенно эффективными являются операции над отдельными символами строки. В Delphi 7 имеется много встроенных процедур обработки строк, выполняющих копирование, выделение подстрок, конкатенацию (объединение). Несмотря на наличие таких процедур, на практике часто необходимо разрабатывать весьма специфические алгоритмы обработки, которые не могут быть эффективно реализованы в Delphi. Некоторые из таких алгоритмов, в которых используются процедуры на встроенном ассемблере, представлены в последующих примерах.

Начнем с примеров работы со строками с завершающим нулем. На практике часто приходится находить размер такой строки. Поместим на главную форму приложения два поля редактирования Edit с именами Edit1 и Edit2, две метки Label и кнопку Button. Исходный текст приложения представлен в листинге 6.33.

Листинг 6.33. Программа, вычисляющая размер строки с завершающим нулем

```
unit sspas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  Tform1 = class(Tform)
```

```
    Button1: Tbutton;
    Edit1  : Tedit;
    Edit2  : Tedit;
    Label1 : Tlabel;
    Label2 : Tlabel;
    procedure Button1Click(Sender: Tobject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  s1: PChar;
  ls1: Integer;

implementation

{$R *.dfm}

function NulTermLen(s1: PChar): Integer; stdcall;
asm
  push    EDI
  mov     EDI, s1
  mov     EDX, EDI

  mov     AL, 0
  cld
@next:
  scasb
  jne     @next
  sub     EDI, EDX
  mov     EAX, EDI
  dec     EAX
  pop     EDI
end;
```

16 Зак. 1061

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  s1 :='Hello from BASM!'#0;
  Edit1.Text := s1;
  ls1 := NulTermLen(s1);
  Edit2.Text := IntToStr(ls1);
end;
end.
```

Строка с завершающим нулем `s1` определена как указатель типа `PChar` в секции объявления переменных, а инициализация выполняется в обработчике нажатия кнопки. Ассемблерная процедура `NulTermLen` подсчитывает размер строки, принимая в качестве входного параметра ее адрес. Для операций сравнения каждого символа строки с нулем используется строковая команда `scasb`. Для ее выполнения требуется, чтобы в регистре `AL` находился символ, который необходимо найти, а в регистре `EDI` — адрес строки. Флаг направления устанавливается в положение 0 для инкремента адресов:

```
   ...
  mov     EDI, s1
   ...
  mov     AL, 0
  cld
@next:
  scasb
  jne     @next
   ...
```

Если нулевой элемент не найден, то переходим к следующей итерации по команде условного перехода. В противном случае вычисляем размер строки как разность значений, находящихся в регистрах `EDI` и `EDX`. Регистр `EDI` содержит адрес, на 1 больший чем адрес обнаруженного нулевого элемента, а регистр `EDX` — адрес строки (указатель на первый элемент). Результирующее значение помещается в регистр `EDI`. Результат, уменьшенный на 1, помещается, как обычно, в регистр `EAX`:

```
sub     EDI, EDX
mov     EAX, EDI
dec     EAX
```

Окно работающего приложения изображено на рис. 6.11.

Рис. 6.11. Окно приложения, отображающего размер строки с завершающим нулем

Рассмотрим еще один пример, в котором подсчитывается количество слов в строке с завершающим нулем. Предположим, что слова отделяются друг от друга символом пробела. На главной форме приложения разместим два поля редактирования `Edit`, кнопку `Button` и две метки статического текста `Label`.

Исходный текст нашего приложения приведен далее в листинге 6.34.

Листинг 6.34. Программа, выполняющая подсчет слов в строке с завершающим нулем

```
unit cwpas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms,  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1  : TEdit;
    Edit2  : TEdit;
    Label1 : TLabel;
    Label2 : TLabel;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  s1: PChar;
  ls1: Integer;
  numWords: Integer;

implementation

{$R *.dfm}

function RetNumWords(s1:PChar; ls1: Integer): Integer; stdcall;
asm
  push    EDI
  mov     EDI, s1
  mov     ECX, ls1
  xor     EDX, EDX
  cmp     BYTE PTR [EDI], ' '
  je      @cont
  inc     EDX
@cont:
  cld
  mov     AL, ' '
@next:
  scasb
  jne     @skip
  cmp     BYTE PTR [EDI], ' '
  jne     @incEDX
@skip:
  loop    @next
  jmp     @exit
@incEDX:
  cmp     BYTE PTR [EDI], 0
  je      @exit
  inc     EDX
  jmp     @next
```

```
@exit:
  pop     EDI
  mov     EAX, EDX
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  s1 := ' The words of this string will be counted!!'#0;
  Edit1.Text := s1;
  ls1 := StrLen(s1);
  numWords := RetNumWords(s1, ls1);
  Edit2.Text := IntToStr(numWords);
end;
end.
```

В этом примере строка с завершающим нулем определена как указатель `PChar`. Инициализация строки происходит в теле обработчика нажатия кнопки. В качестве разделителя слов в этой строке используется пробел. Автор специально выбрал строку с произвольным расположением слов и произвольным расположением разделителей для усложнения задачи. Параметрами процедуры `RetNumWords` служат адрес строки и ее размер. Возвращаемой величиной является количество слов в строке. Команды:

```
mov     EDI, s1
mov     ECX, ls1
xor     EDX, EDX
```

загружают адрес строки в регистр `EDI`, а размер строки — в регистр `ECX`. Регистр `EDX` используется в качестве счетчика слов и в начале процедуры обнуляется. Следующий фрагмент кода отрабатывает ситуацию, при которой пробелы в начале строки (если они есть) будут пропущены и учитываться не будут:

```
   ...
  cmp     BYTE PTR [EDI], ' '
  je      @cont
  inc     EDX
@cont:
   ...
```

После этого процедура выполняет операцию сравнения текущего символа строки с символом пробела:

```
   ...
  cld
  mov     AL, ' '
@next:
  scasb
  jne     @skip
  cmp     BYTE PTR [EDI], ' '
  jne     @incEDX
@skip:
  loop    @next
   ...
```

Для выполнения сравнения нужно поместить в регистр AL символ пробела и командой scasb просматривать строку. Поскольку мы рассматриваем самый общий случай размещения слов и разделителей, то не исключен вариант, когда пробелов между словами будет несколько. Для корректной отработки такой ситуации служит команда:

```
cmp     BYTE PTR [EDI], ' '
```

которая передает управление на метку @next, пока встречается символ пробела. Если найден первый символ строки, не равный пробелу, произойдет переход на метку @incEDX, и содержимое счетчика слов увеличится на 1. Число итераций не превысит содержимого счетчика ECX. При выходе из цикла loop регистр EDX содержит количество слов в строке. Это значение возвращается в регистре EAX в основную программу.

В обработчике нажатия кнопки происходит инициализация строки s1, вычисление ее размера ls1 и вызов процедуры RetNumWords. Вычисленное значение ls1 вместе с адресом строки передается процедуре в качестве параметров:

```
ls1 := StrLen(s1);
numWords := RetNumWords(s1, ls1);
```

Окно работающего приложения изображено на рис. 6.12.

Рис. 6.12. Окно приложения, отображающего количество слов в строке

Еще одно замечание. В качестве разделителя слов может служить любой другой символ, например символ табуляции. В этом случае достаточно заменить символ пробела в соответствующих командах программы на нужный символ-разделитель.

Наш следующий пример связан с преобразованием и обработкой строк различного типа. Это одна из довольно сложных проблем, с которыми сталкивается программист при написании Delphi-приложений. Дело в том, что в Delphi многие функции обработки текстовых и символьных данных выполняются с использованием длинных строк (ANSI String), о которых было упомянуто ранее. Между тем, Windows в качестве стандарта использует строки с завершающим нулем. Следовательно, необходимо преобразовать длинную строку в строку с завершающим нулем.

Можно задать вопрос: а не будет ли проще работать напрямую с длинными строками без этих преобразований? Если только работать в Delphi и не использовать мощный арсенал функций WIN API и ассемблера, то да. Но, скорее всего, такое ограничение не обрадует программиста, поскольку вряд ли более-менее серьезное приложение обойдется без подобного инструментария.

Хотя разработчики Delphi сделали максимум возможного для совместимости этих строк, отличия все же остаются, поэтому приходится использовать определенные ухищрения для конвертации строк и обработки данных.

В качестве примера разработаем программу, в которой будет анализироваться строка, где разделителями слов являются символы пробела. Все такие символы заменим для наглядности на символ "+". На главной форме приложения разместим два поля редактирования `Edit`, кнопку `Button` и две метки статического текста `Label`. Одно поле редактирования принимает исходную строку, а другое — выполняет вывод модифицированной строки. Исходный текст приложения приведен в листинге 6.35.

Листинг 6.35. Программа, выполняющая поиск и замену разделителей в строке определенным символом

```
unit rcpas;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1  : TEdit;
    Label1 : TLabel;
    Edit2  : TEdit;
    Label2 : TLabel;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  lensrc: Integer;
  buf: PChar;

implementation

{$R *.dfm}

procedure ReplaceSpace(sx: PChar; stl: Integer); stdcall;
asm
  push    EDI

  mov     EDI, DWORD PTR sx
```

```
  mov     ECX, DWORD PTR stl
  cld
  mov     AL, ' '
@next:
  scasb
  jne     @skip
  mov     BYTE PTR [EDI-1], '+'
@skip:
  loop    @next
  pop     EDI
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  lensrc := Edit1.GetTextLen;
  Inc(lensrc);
  GetMem(buf, lensrc);
  Edit1.GetTextBuf(buf, lensrc);
  ReplaceSpace(buf, lensrc);
  Edit2.Text := StrPas(buf);
  FreeMem(buf, lensrc);
end;
end.
```

Строки в полях редактирования имеют тип ANSI String, а обрабатывать их очень удобно, предварительно преобразовав в строки с завершающим нулем. В нашем примере обработка строки с завершающим нулем выполняется ассемблерной процедурой `ReplaceSpace`, которая принимает в качестве параметров указатель на строку типа `PChar` и размер строки.

Предварительное преобразование длинной строки в строку с завершающим нулем выполняется в основной программе:

```
lensrc := Edit1.GetTextLen;
Inc(lensrc);
GetMem(buf, lensrc);
Edit1.GetTextBuf(buf, lensrc);
```

Первое что мы делаем — это получаем размер длинной строки при помощи процедуры `GetTextLen`. Следующим оператором увеличиваем полученный

размер на 1. Если мы планируем работать со строкой типа PChar (строка с завершающим нулем в Delphi), то это необходимо для резервирования места, где будет находиться символ окончания строки, т. е. 0.

Как правило, преобразование одного типа в другой выполняется с помещением исходного значения переменной в буфер памяти. Результат преобразования обычно помещается в эту же область памяти. Поэтому вызов GetMem выделяет буфер памяти для хранения строки размером lensrc. Наконец, GetTextBuf помещает строку из поля редактирования в буфер. После этого в области памяти buf будет находиться строка с завершающим нулем. Последующие преобразования ассемблерной процедурой ReplaceSpace выполняются уже над строкой типа PChar.

Проанализируем работу процедуры ReplaceSpace. Она принимает в качестве параметров адрес строки с завершающим нулем и ее размер. В начале процедуры загружаем адрес строки в регистр EDI, а ее размер — в регистр ECX, который будем использовать в качестве счетчика цикла. Для организации поиска символа пробела воспользуемся знакомой нам командой scasb:

```
   ...
  cld
  mov     AL, ' '
@next:
  scasb
  jne     @skip
   ...
```

Установим флаг направления так, чтобы адрес инкрементировался в каждой итерации. Если символ пробела найден, то необходимо его заменить на "+". Поскольку после выполнения команды scasb содержимое регистра EDI увеличилось на 1, то записать символ "+" необходимо в ячейку памяти с адресом [EDI-1]:

```
mov     BYTE PTR [EDI-1], '+'
```

После замены всех пробелов нам необходимо вывести результат в поле редактирования Edit2. Для этого выполним преобразование строки типа PChar в ANSI String и присвоим полученный результат свойству Text компонента Edit2. Это выполняет оператор:

```
Edit2.Text := StrPas(buf);
```

И, наконец, последней командой обработчика нажатия кнопки освободим буфер памяти:

```
FreeMem(buf, lensrc);
```

Окно работающего приложения изображено на рис. 6.13.

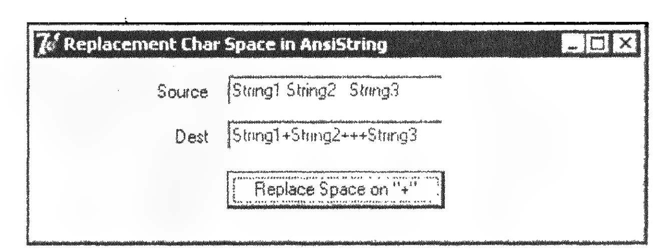


Рис. 6.13. Окно приложения, выполняющего замену символов в строке типа ANSI String

Среда разработки Delphi имеет много встроенных процедур обработки строк, причем довольно эффективных. Может возникнуть вопрос: зачем изобретать велосипед и писать процедуры на ассемблере, если аналогичные уже написаны на Delphi и готовы к использованию?

Не случайно выбраны последние два варианта программ в качестве примеров. Ассемблерные процедуры, выполняющие "точечные" преобразования в строке, такие как замена, поиск символов и подстрок (фрагментов строк), всегда превосходят свои аналоги на языках высокого уровня. Процедуры на ассемблере в этих случаях чрезвычайно эффективны и выполняются намного быстрее; не важно, с каким компилятором вы работаете и какую среду предпочитаете — Delphi 7 или Visual C++ .NET. Чтобы убедиться в этом, читатель может переписать приведенный последний пример, используя только высокоуровневые функции Delphi для обработки строк, и дизассемблировать программный код.

Описание встроенного ассемблера Delphi 7 было бы неполным, если бы мы не рассмотрели еще одну замечательную возможность, предоставляемую этим средством. Ассемблерная процедура может напрямую обращаться к функциям WIN API операционной системы Windows. Это позволяет расширить возможности программы за счет манипулирования мощными функциями библиотеки WIN API. Как можно вызвать такие функции из ассемблерного кода?

Вспомним, что для вызова функций WIN API необходимо передать параметры в вызываемую функцию через стек в порядке, обратном их следованию в списке аргументов функции. В этом случае самый правый параметр функции помещается в стек первым. Кроме того, почти все функции

WIN API, как мы знаем, используют соглашения директивы stdcall. Это значит, что при возврате управления вызывающей программе вызываемая функция сама должна освобождать стек.

Рассмотрим пример использования трех функций WIN API: CreateFile, WriteFile, CloseHandle. Требуется записать в файл с именем TEXTOUT строчку текста. Поместим на главной форме приложения два поля редактирования Edit с именами Edit1 и Edit2, две метки Label и кнопку Button. Текст для записи в файл возьмем из поля редактирования Edit1. В поле редактирования Edit2 будет отображаться количество байт, записанных в файл. Процесс обработки текста, введенного пользователем, выполняется в обработчике нажатия кнопки, которую мы предварительно разместим на главной форме приложения.

Исходный текст программы приведен в листинге 6.36.

Листинг 6.36. Использование функций WIN API в блоке asm-end для записи строки текста в файл

```
unit apiuse;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
  Path: PChar;

  sTitle, crText, wrText, clText: PChar;
  pBuf: PChar;
  lenBuf: Integer;

  hFile: DWORD;
  bufWritten: DWORD;
begin
  sTitle := 'WIN API USE'#0;
  crText := 'File Creating Error!'#0;
  wrText := 'File Writing Error!'#0;
  clText := 'File Closing Error!'#0;

  Path := 'TEXTOUT'#0;

  lenBuf := Edit1.GetTextLen;
  inc(lenBuf);
  GetMem(pBuf, lenBuf);

  Edit1.GetTextBuf(pBuf, lenBuf);

  asm
    push    0
    push    FILE_ATTRIBUTE_NORMAL
    push    CREATE_ALWAYS
    push    0
    push    0
    push    GENERIC_WRITE
    push    DWORD PTR Path
    call    CreateFile
```

```
  cmp     EAX, INVALID_HANDLE_VALUE
  je      @crMes
  mov     DWORD PTR hFile, EAX

  push    0
  lea     ESI, DWORD PTR bufWritten
  push    ESI
  dec     DWORD PTR lenBuf
  push    DWORD PTR lenBuf
  push    DWORD PTR pBuf
  push    hFile
  call    WriteFile
  cmp     EAX, 1
  je      @goClose
  jmp     @wrMes

@goClose:
  push    hFile
  call    CloseHandle
  cmp     EAX, 1
  jne     @clMes
  jmp     @ex
@crMes:
  push    MB_OK
  push    DWORD PTR sTitle
  push    DWORD PTR crText
  jmp     @cont
@wrMes:
  push    MB_OK
  push    DWORD PTR sTitle
  push    DWORD PTR wrText
  jmp     @cont
@clMes:
  push    MB_OK
  push    DWORD PTR sTitle
  push    DWORD PTR clText
```

```
  @cont:
    push    0
    call    MessageBox
  @ex:
  end;
  FreeMem(pBuf, lenBuf);
  Edit2.Text := IntToStr(bufWritten);
end;

end.
```

Проанализируем исходный текст программы. В секции `var` объявлены следующие переменные:

- `Path` определяет имя создаваемого файла;
- `pBuf` — указатель на буфер памяти, данные из которого будут записаны в файл;
- `lenBuf` — размер буфера памяти;
- `hFile` — дескриптор файла, в который записываются данные;
- `bufWritten` содержит количество записанных байт;
- `sTitle`, `crText`, `wrText`, `clText` — заголовок окна и сообщения, указывающие тип возможной ошибки при выполнении операции с файлом.

Первое, что делает программа, — помещает введенный в поле редактирования текст в буфер памяти. Это выполняется группой операторов:

```
lenBuf := Edit1.GetTextLen;
inc(lenBuf);
GetMem(pBuf, lenBuf);
Edit1.GetTextBuf(pBuf, lenBuf);
```

Первый оператор сохраняет размер строки из поля редактирования в переменной `lenBuf`. Второй по порядку оператор резервирует место для нулевого символа через инкремент переменной `lenBuf`. Оператор:

```
GetMem(pBuf, lenBuf)
```

позволяет выделить память, поместив в указатель `pBuf` адрес первой ячейки выделенной области. Наконец, с помощью оператора:

```
Edit1.GetTextBuf(pBuf, lenBuf)
```

текст из поля редактирования Edit1 помещается в буфер памяти. Для записи текстовой строки в файл вначале создадим этот файл. Это выполняется с помощью функции CreateFile:

```
push    0
push    FILE_ATTRIBUTE_NORMAL
push    CREATE_ALWAYS
push    0
push    0
push    GENERIC_WRITE
push    DWORD PTR Path
call    CreateFile
mov     DWORD PTR hFile, EAX
```

Параметры вызова хорошо описаны в документации по WIN API и многочисленных литературных источниках, поэтому подробно останавливаться на этом не будем. Хочется только заметить, что все системные константы Windows поддерживаются средой Delphi. Именно поэтому такая команда, как:

```
push    GENERIC_WRITE
```

ошибки компилятора не вызывает. Все параметры, передаваемые через стек, являются 32-разрядными. Все функции WIN API возвращают результат в регистре EAX. Для анализа содержимого регистра EAX необходимо выполнить команды:

```
cmp     EAX, INVALID_HANDLE_VALUE
je      @crMes
```

Если в EAX возвращается константа INVALID_HANDLE_VALUE, то происходит выход из ассемблерного блока программы. При этом отображается соответствующее сообщение через вызов другой функции WIN API MessageBox. Если удалось создать файл, то в регистре EAX возвращается целочисленный дескриптор файла, который используется при последующих файловых операциях.

На следующем этапе записываем данные из буфера памяти в файл с дескриптором hFile с помощью функции WriteFile. Функция WriteFile возвращает значение типа bool в регистре EAX, сигнализирующее об успешности выполнения операции записи. Булевым значениям в ассемблере соответствует 1 или 0. В зависимости от содержимого EAX выполняется либо

закрытие файла функцией WIN API CloseHandle, либо выход из блока asm-end программы с выводом соответствующего сообщения на экран:

```
   ...
  call    WriteFile
  cmp     EAX, 1
  je      @goClose
  jmp     @wrMes
@goClose:
  push    hFile
  call    CloseHandle
   ...
```

После выполнения записи в файл необходимо его закрыть с помощью функции CloseHandle. Если операция выполнена успешно, то в рабочем каталоге проекта будет находиться файл TEXTOUT с записанной строкой. По завершению записи в файл необходимо освободить буфер памяти, на который указывает pBuf:

```
FreeMem(pBuf, lenBuf);
```

Количество записанных байт содержится в переменной bufWritten и отображается оператором

```
Edit2.Text := IntToStr(bufWritten);
```

Результат работы приложения изображен на рис. 6.14.

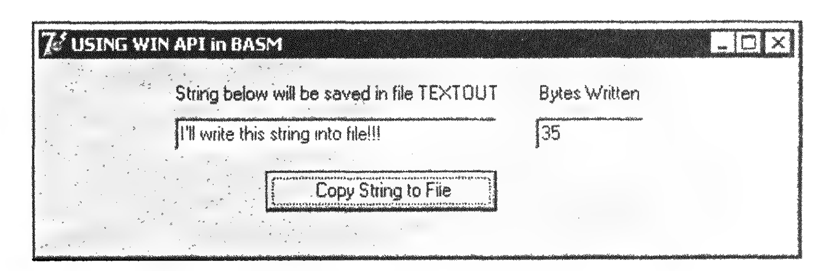


Рис.6.14. Окно приложения, демонстрирующего применение функций WIN API

На этом анализ работы встроенного ассемблера Delphi можно закончить. Далее мы перейдем к рассмотрению встроенного ассемблера Visual C++ .NET.

6.8. Применение встроенного ассемблера в Microsoft Visual C++ .NET

Среда разработки Microsoft Visual C++ .NET включает в себя мощнейшие средства поддержки программирования на языке ассемблера. Любой блок ассемблерного кода, встречающийся в программе, должен начинаться с ключевого слова _asm и заключаться в фигурные скобки, как, например:

```
_asm {
      mov     EAX, val1
      sub     EAX, EBX
     }
```

Можно применять альтернативную форму записи команд ассемблера в строку. Предыдущий фрагмент кода в этом случае будет выглядеть так:

```
_asm mov    EAX, val1
_asm sub    EAX, EBX
```

Допускается и третья форма написания ассемблерного кода — в одну строку:

```
_asm mov    EAX, val1 _asm sub EAX, EBX
```

Встроенный ассемблер C++ .NET является весьма эффективным средством оптимизации программ. Немного о терминологии. В языке C++ принято для определения отдельных подпрограмм использовать термин "функция", независимо от того, возвращает она результат или нет. В дальнейшем по тексту мы будем придерживаться этого определения.

У программистов, использующих ассемблер MASM, сразу может возникнуть вопрос: в какой мере среда разработки C++ .NET поддерживает синтаксис этого языка. Многие конструкции MASM, такие как DB, DW, DD, DQ, DF или операторы DUP и THIS, не поддерживаются. Встроенный ассемблер не поддерживает и такие директивы, как STRUC, RECORD, WIDTH, MASK.

Операторы ассемблера LENGTH, SIZE или TYPE ограничены в применении в Visual C++ .NET. Их нельзя применить с оператором DUP, поскольку для определения данных директивы DB, DW, DD, DQ и DF не используются. Однако их можно использовать для определения размеров переменных следующим образом:

- оператор LENGTH возвращает число элементов массива или единицу для обычных переменных;

- оператор SIZE возвращает размер переменной языка С или С++;
- оператор TYPE возвращает размер переменной. Если переменная указывает на массив, то этот оператор возвращает размер одного элемента массива.

Например, если в программе используется массив целых чисел из 20-ти элементов, объявленный как:

```
int iarray[20],
```

то результат применения этих операторов ассемблера будет выглядеть так, как показано в табл. 6.2.

Таблица 6.2. *Соответствие операторов ассемблера операторам С++*

Оператор _asm	Аналог оператора в С	Размер
LENGTH iarray	sizeof(iarray)/sizeof(iarray[0])	20
SIZE iarray	sizeof(iarray)	80
TYPE iarray	sizeof(iarray[0])	4

Комментарии в исходном тексте программы отделяются от операторов, как и в MASM, точкой с запятой, например,

```
_asm {
    mov     EAX, val1          ; Это комментарий к первой строке
    sub     EAX, EBX           ; Это комментарий ко второй строке
   }
```

Поскольку команды встроенного ассемблера чередуются с операторами С++, то они могут ссылаться на структуры и переменные, используемые в С++ .NET. В ассемблерном блоке могут использоваться разнообразные элементы языка С++:

- символы, включая метки, переменные и имена функций;
- константы, в том числе символьные и строковые;
- макросы и директивы препроцессора;
- комментарии в С-стиле (/**/ и //);
- typedef имена, используемые обычно вместе с операторами PTR и TYPE или для доступа к элементам объединения (union) или структуры (structure).

Внутри ассемблерного блока можно определять целочисленные константы, соответствующие правилам, принятым как в C++, так и в ассемблере. Например, символ пробела может быть записан и как 0x20, и как 20h.

Допускается использование директивы define для определения констант. Такое определение будет действовать как в ассемблерном блоке, так и в программе на C++.

Что касается применения операторов C++ в ассемблерных блоках, то здесь есть некоторые нюансы. В блоке _asm нельзя использовать специфические для языка C++ операторы. В то же время некоторые операторы совершенно по-разному трактуются в ассемблере и в C++. Например, оператор квадратных скобок [] в C++ используется для указания размеров массива. Во встроенном ассемблере этот же оператор применяется для индексирования доступа к переменным. Если неправильно применять операторы в блоке _asm, то обнаружить ошибки в программе будет очень трудно.

Привем пример правильного и неправильного использования оператора квадратных скобок. Для этого разработаем приложение на C++. NET.

Поместим на главную форму приложения три поля редактирования Edit, кнопку Button и три метки статического текста Label. Основная программа будет содержать функцию, написанную на встроенном ассемблере, и обработчик нажатия кнопки. В обработчике будет происходить визуализация вычислений, выполненных на ассемблере. Для тестирования возьмем 5-элементный массив целых чисел. Попробуем заменить в нем элемент с индексом 3 (четвертый по порядку) на число −115. Число выбрано произвольно. Замену выполним двумя способами, в обоих случаях будем блок _asm.

Для форматирования вывода и отображения элементов массива в полях редактирования понадобятся переменные типа CString, которые мы свяжем с элементами Edit. Полю Edit1 (метка Original) присвоим переменную iOrigin, полю Edit2 (метка Correct) присвоим iAsmCorr и полю Edit3 (метка Wrong) — iAsmWrong. При нажатии на кнопку в полях редактирования будут отображены элементы исходного массива (Edit1), элементы корректно преобразованного массива (Edit2) и элементы неправильно преобразованного массива (Edit3). Исходный текст обработчика нажатия кнопки, в котором выполняется обработка массива, представлен в листинге 6.37.

Листинг 6.37. Программа, демонстрирующая применение оператора скобки в _asm блоке

```
   ...
#include <string.h>
#define  NUM_BYTES 4
   ...
void CUSINGOPERATORSIN_ASMBLOCKDlg::OnBnClickedButton1()
```

```
{
  // TODO: Add your control notification handler code here

  int arr[5] = {4, 0, 9, -7, 50};
  int arrw[5], arrc[5];

  memcpy(arrw, arr, NUM_BYTES * 5);
  memcpy(arrc, arr, NUM_BYTES * 5);

  int* parr = arr;
  int isize = sizeof(arr) / 4;
  int cnt;
  CString stmp;

  stmp.Empty;
  for (cnt = 0; cnt < isize; cnt++)
  {
    stmp.Format("%d", *parr);
    iOrigin = iOrigin + "  " + stmp;
    parr++;
  };

  parr = arrc;
  _asm {
        mov     EAX, -115
        mov     arrc[3 * TYPE int], EAX
       };

  stmp.Empty;
  for (cnt = 0; cnt < isize; cnt++)
  {
    stmp.Format("%d", *parr);
    iAsmCorr = iAsmCorr + "  " + stmp;
    parr++;
  };
```

```
  parr = arrw;
  _asm {
        mov EAX, -115
        mov arrw[3], EAX
       };

  stmp.Empty;
  for (int cnt = 0; cnt < isize; cnt++)
  {
    stmp.Format("%d", *parr);
    iAsmWrong = iAsmWrong + "  " + stmp;
    parr++;
  };
  UpdateData(FALSE);
};
```

Результат работы программы изображен на рис. 6.15.

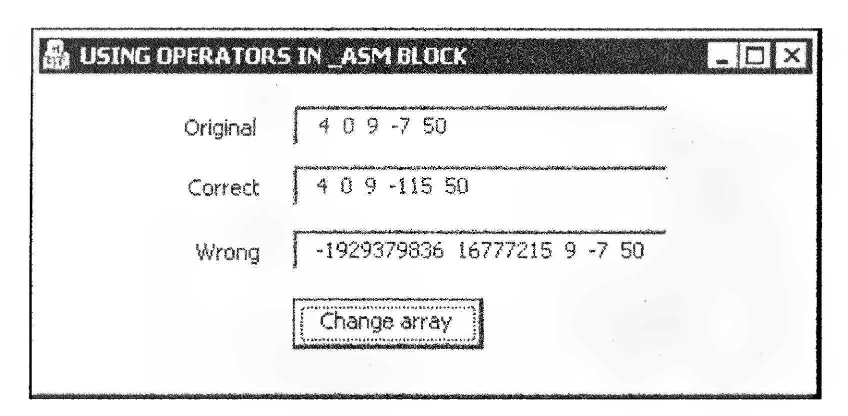


Рис. 6.15. Окно приложения, демонстрирующего правильное и неправильное использование операторов C++

Проведем анализ программного кода обработчика. В начале программы создается две копии исходного массива с помощью операторов:

```
memcpy(arrw, arr, NUM_BYTES * 5);
memcpy(arrc, arr, NUM_BYTES * 5);
```

Здесь копируются байты, поэтому последний параметр функции memcpy равен количеству байт в массивах. Прототип функции определен в файле

string.h, поэтому в текст программы в раздел деклараций необходимо добавить строчку:

```
#include <string.h>
```

В обработчике нажатия кнопки присутствуют три цикла for, которые подготавливают буферы переменных iOrigin, iAsmCorr и iAsmWrong для вывода элементов массивов в поля редактирования. Рассмотрим подробно, что происходит с массивами arrc и arrw при попытке заменить в них четвертый элемент. Для массива arrc запись числа −115 выполняется следующим образом:

```
parr = arrc;                          // инициализация указателя
_asm {
      mov EAX, -115                   // запись числа в регистр EAX
      mov arrc[3 * TYPE int], EAX     // запись содержимого EAX по адресу
                                      // элемента с индексом 3 (правильно!)
     };
```

После выполнения этих команд в четвертом элементе массива находится −115. Другая ситуация с массивом arrw. Запись по адресу четвертого элемента выполняется операторами:

```
parr = arrw;
_asm {
      mov EAX, -115
      mov arrw[3], EAX                // НЕПРАВИЛЬНАЯ КОМАНДА!
     };
```

После выполнения этих команд будут перезаписаны четыре байта памяти, начиная с элемента с индексом 3. Поскольку 4-й байт является последним для первого элемента массива, а последующие (с 5 по 7-й) захватывают второе число в массиве, то в результате содержимое первых двух элементов массива будет разрушено, что видно из рис. 6.15.

Ситуации, подобные этой, могут встретиться при работе со встроенным ассемблером C++ .NET, поэтому надо тщательно отслеживать все преобразования с применением ассемблера.

Как уже было сказано, в ассемблерном блоке можно ссылаться на любые символы языка C++, хотя и существуют некоторые ограничения:

- в каждой ассемблерной команде может содержаться ссылка только на один символ (переменную, функцию или метку). Для использования нескольких символов в одной команде необходимо, чтобы все они применялись в выражениях типа LENGTH, TYPE и SIZE;

- функции, на которые ссылаются команды ассемблерного блока, должны быть заранее объявлены в программе, иначе компилятор не сможет отличить ссылку на функцию от метки;
- в ассемблерном блоке нельзя использовать символы C++, которые схожи по написанию с директивами MASM;
- в ассемблерном блоке не распознаются структуры и объединения.

Наиболее ценной особенностью встроенного ассемблера C++ .NET является его способность распознавать и использовать переменные языка C++. Если в модуле, где используется ассемблер, определены, например, переменные `val1` и `val2`, то следующая ссылка в ассемблерном блоке будет корректной:

```
_asm {
     mov     EAX, val1
     add     EAX, val2
    }
```

Как известно, функции в языке C++ возвращают результат в основную программу, используя оператор `return`. Например, следующая функция (назовем ее `MulInts`) возвращает в основную программу значение `i1 * i2 + 100` (листинг 6.38).

Листинг 6.38. Функция, возвращающая результат с помощью оператора return

```
int CReturnValueinregisterEAXwithinlineassemblerDlg::MulInts(int i1,
                                                          int i2)
{
  int valMul;
  _asm {
       mov     EAX, i1
       mov     EBX, i2
       mul     EBX
       xchg    EAX, EDX
       add     EDX, 100
       mov     valMul, EDX
  };
  return valMul;
}
```

Встроенный ассемблер позволяет обходиться без оператора `return` при возвращении результата, используя для этого регистр EAX. Та же самая функция `MulInts` при определенных изменениях исходного текста может использовать такую возможность (листинг 6.39).

Листинг 6.39. Функция, возвращающая результат в регистре EAX

```
int CReturnValueinregisterEAXwithinlineassemblerDlg::MulInts(int i1,
                                                             int i2)
{
  _asm {
        mov     EAX, i1
        mov     EBX, i2
        mul     EBX
        add     EAX, 100
  };
}
```

Несмотря на то, что функция не возвращает результат через оператор `return`, компилятор не выдаст сообщение об ошибке.

При написании ассемблерного кода нет необходимости сохранять регистры EBX, ESI и EDI. Однако если регистры используются в программе, то компилятор будет сохранять их при вызове функции и автоматически восстанавливать после выхода из нее. При частом вызове такой функции может несколько снизиться быстродействие.

Если ваша программа использует команды `cld` или `std`, то необходимо восстанавливать значение флага направления при выходе из функции.

После теории можно перейти к демонстрации возможностей встроенного ассемблера на примерах конкретных программ обработки числовых и текстовых данных. Начнем с примера вычисления суммы двух целых чисел `i1` и `i2`. Создадим каркас приложения на базе диалогового окна и выполним некоторые дополнительные действия. Разместим на главной форме окна три поля редактирования `Edit` (для ввода двух целых чисел и вывода результата) и одну кнопку `Button`. Вычисление суммы чисел будет выполнять ассемблерная функция `SumTwoInts`.

Как известно, элементам управления в C++ .NET можно поставить в соответствие переменные того или иного типа. Манипуляции с элементами управления в этом случае могут выполняться через их переменные. Свяжем с элементом `Edit1` переменную целого типа `i_I1`, а с элементом управления

Edit2 — переменную i_I2. Элементу управления Edit3 поставим в соответствие переменную i_I1I2. После этого можно использовать все эти переменные в выражениях C++. Вычисление суммы выполняет функция SumTwoInts, а ввод-вывод осуществляет обработчик нажатия кнопки. Фрагменты программного кода приведены в листинге 6.40.

Листинг 6.40. Вычисление суммы двух целых чисел и вывод результата

```
int CSummationofTwoIntegersDlg::SumTwoInts(int i1, int i2)
{
  _asm {
        mov     EAX, i1
        add     EAX, i2
  }
};

void CSummationofTwoIntegersDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  UpdateData(TRUE);
  i_I1I2 = SumTwoInts(i_I1, i_I2);
  UpdateData(FALSE);
}
```

При написании исходного текста функции SumTwoInts мы воспользовались возможностью передачи результата вычислений в регистре EAX. При передаче параметров не нужно явно указывать размер операндов. По умолчанию компилятор считает размер целочисленного операнда равным 4 байтам, поэтому команды блока _asm транслируются корректно. Первый оператор обработчика нажатия кнопки:

```
UpdateData(TRUE);
```

обновляет значения переменных i_I1 и i_I2 в соответствии с текущими значениями элементов управления Edit1 и Edit2. Другими словами, если пользователь ввел в окне Edit1 значение 34, а в окне Edit2 — 67, то после выполнения оператора UpdateData(TRUE) эти значения будут присвоены

переменным i_I1 и i_I2 соответственно. Выполнение следующего оператора:

```
i_I1I2 = SumTwoInts(i_I1, i_I2);
```

позволяет сохранить результат выполнения функции SumTwoInts в переменной i_I1I2. И, наконец, оператор:

```
UpdateData(FALSE);
```

обновляет текущее содержимое элементов управления в соответствии со значениями их переменных.

Окно работающего приложения изображено на рис. 6.16.

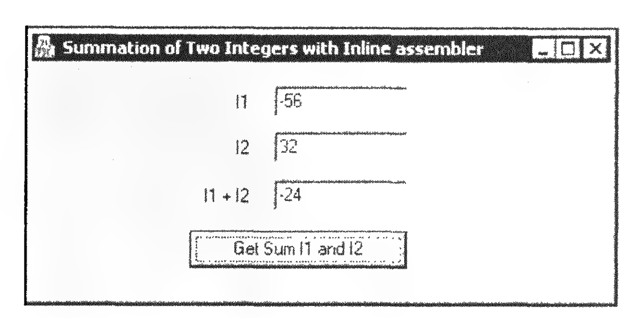


Рис. 6.16. Окно приложения, выполняющего суммирование целых чисел

Следующий пример — вычисление суммы элементов массива вещественных чисел. Помимо демонстрации работы математического сопроцессора здесь будет показано, как передавать результат выполнения функции через указатель.

На главной форме приложения разместим два поля редактирования Edit и одну кнопку Button. В первом поле редактирования Edit1 выведем весь массив вещественных чисел, а во втором поле Edit2 будет выведен результат вычислений.

Свяжем с элементами управления Edit1 и Edit2 две переменные — s_Array и f_Summa соответственно. Переменной s_Array присвоим строковый тип CString, а переменной f_Summa — вещественный float.

Вычисление суммы элементов массива вещественных чисел выполним с помощью функции sumReals. В качестве параметров эта функция принимает адрес массива и его размер. Исходный текст этой функции и обработчика нажатия кнопки приведен в листинге 6.41.

Листинг 6.41. Вычисление суммы элементов массива вещественных чисел и вывод результата

```
float* CSummaofRealsDlg::sumReals(float* farray, int lf)
{
  float fsum;
  _asm {
        mov     ESI, farray
        mov     ECX, lf
        dec     ECX
        finit
        fldz
        fld     [ESI]
      next:
        add     ESI, 4
        fadd    [ESI]
        loop    next
        fstp    fsum
        fwait
        lea     EAX, fsum
  };
}

void CSummaofRealsDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  float farray[] = {2.4, 5.9, -4.12, 3.12,-8.45};

  int fsize = sizeof(farray)/4;
  CString stmp;

  UpdateData(TRUE);
  stmp.Empty;
  for (int cnt = 0; cnt < fsize; cnt++)
  {
    stmp.Format("%.2f", farray[cnt]);
    s_Array = s_Array + "  " + stmp;
  };
```

```
  f_Summa = *sumReals(farray, fsize);
  UpdateData(FALSE);
}
```

Как уже упоминалось, при вызове функции нет необходимости сохранять регистр ESI. Поэтому в самом начале функции sumReals загружаем адрес массива farray в регистр ESI, а его размер — в регистр ECX:

```
mov     ESI, farray
mov     ECX, lf
```

Адресом массива farray является указатель на первый элемент этого массива. Содержимое регистра ECX используется для организации вычисления в цикле. Самое первое значение элемента массива загружается в вершину стека сопроцессора с помощью команды:

```
fld     [ESI]
```

В каждой операции цикла выполняется продвижение адреса к следующему элементу массива и прибавляется значение последующего элемента к содержимому вершины стека:

```
next:
  add     ESI, 4
  fadd    [ESI]
  loop    next
```

Результат суммирования сохраняется в локальной переменной fsum:

```
fstp    fsum
```

Последняя команда ассемблерной функции:

```
lea     EAX, fsum
```

загружает адрес переменной fsum в регистр EAX. Этот адрес функция возвращает в основную программу.

В обработчике нажатия кнопки мы используем строковые переменные для преобразования числовых значений в текстовый формат. Отложим рассмотрение строк до следующих примеров, а сейчас проанализируем один оператор:

```
f_Summa = *sumReals(farray, fsize);
```

В этом операторе `f_Summa` — переменная вещественного типа, а функция `sumReals` возвращает адрес. В этом случае для получения значения по адресу используется оператор раскрытия ссылки "`*`" перед адресным выражением, в нашем случае перед идентификатором функции.

Первый параметр, передаваемый в функцию `sumReals`, — адрес массива. Он может быть представлен в альтернативной форме. Как мы знаем, адрес массива указывает на первый элемент, поэтому рассмотренный оператор можно записать и в другой форме:

```
f_Summa = *sumReals(&farray[0], fsize);
```

где для представления первого параметра в корректной форме используется оператор получения адреса "`&`".

Окно работающего приложения изображено на рис. 6.17.

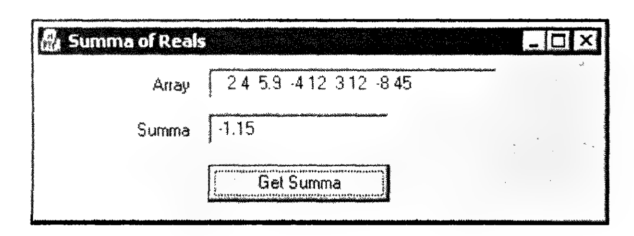


Рис. 6.17. Окно приложения, выполняющего подсчет суммы элементов массива вещественных чисел

В следующих примерах рассмотрим реализацию простых алгоритмов сортировки и поиска максимального элемента. На этих примерах постараемся оценить эффективность встроенного ассемблера для оптимизации программ.

Вначале рассмотрим механизм поиска максимального элемента в массиве целых чисел. Используя стандартный мастер приложения, разработаем каркас приложения на основе диалогового окна.

Разместим на главной форме приложения два поля редактирования `Edit` и кнопку `Button`. Поиск максимального элемента целочисленного массива будет выполнять функция `findMax`. В качестве параметров функция принимает адрес массива и его размер. Результатом выполнения функции является значение максимального элемента в массиве. Фрагменты кода функции `findMax` и обработчика нажатия кнопки приведены в листинге 6.42.

Листинг 6.42. Вычисление максимального значения в массиве целых чисел и вывод результата

```
int CFindMaximumIntegerinArrayDlg::findMax(int* pi1, int si1)
```

```
{
  int maxVal;
  _asm {
        mov     EDI, pi1
        mov     ECX, si1
        mov     EDX, [EDI]
        mov     maxVal, EDX
      next:
        add     EDI, 4
        mov     EAX, [EDI]
        cmp     EAX, maxVal
        jl      no_change
        push    EAX
        pop     maxVal
      no_change:
        loop    next
      ex:
        mov     EAX, maxVal
  };
}

void CFindMaximumIntegerinArrayDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  CString s1;
  int i1[] = {4, -6, 9, -7, 32, -90, 123};
  int si1 = sizeof(i1)/4;
  s1.Empty;
  for (int cnt = 0; cnt < si1; cnt++)
  {
    s1.Format("%d", i1[cnt]);
    s_Array = s_Array + " " + s1;
  };
  i_Max = findMax(i1, si1);
  UpdateData(FALSE);
}
```

Рассмотрим алгоритм работы функции `findMax`. Первый элемент массива считается максимальным и сравнивается в цикле с последующими элемен-

тами. Если следующий элемент больше текущего максимума, то максимальным элементом становится он. Цикл повторяется до тех пор, пока не будет достигнут конец массива. Команды:

```
mov     EDI, pil
mov     ECX, sil
```

загружают в регистры EDI и ECX, соответственно, указатель массива и его размер. Сравнение текущего значения максимума и элемента массива, а также выбор нового значения в качестве максимума выполняется группой команд:

```
  ...
next:
  add     EDI, 4
  mov     EAX, [EDI]
  cmp     EAX, maxVal
  jl      no_change
  push    EAX
  pop     maxVal
  ...
```

Функция возвращает значение максимума, как обычно, в регистре EAX:

```
mov     EAX, maxVal
```

Обработчик нажатия кнопки ничего особенного не выполняет. Полученное значение максимума присваивается переменной i_Max, соответствующей элементу управления Edit2, и выводится на экран.

Окно работающего приложения изображено на рис. 6.18.

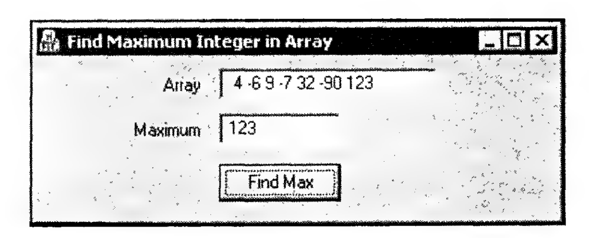


Рис. 6.18. Окно приложения, выполняющего поиск максимального элемента в массиве целых чисел

Было бы интересно сравнить программы поиска максимума как с использованием ассемблерной функции, так и на "чистом" C++.

Ассемблерный вариант у нас есть, поэтому разработаем такую же программу на языке высокого уровня. Программный код обработчика нажатия кнопки приведен далее в листинге 6.43. Фрагмент кода, выполняющий поиск максимального значения, выделен жирным шрифтом.

Листинг 6.43. Фрагмент программы с использованием операторов C++, выполняющий поиск максимального элемента

```
void CFindMaxIntwithCDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  CString s1;
  int i1[] = {4, -6, 9, -7, 32, -90, 123};
  int si1 = sizeof(i1)/4;

  // Вывод элементов массива на экран

  s1.Empty;
  for (int cnt = 0; cnt < si1; cnt++)
  {
    s1.Format("%d", i1[cnt]);
    s_Array = s_Array + " " + s1;
  };

  // Поиск максимального элемента в массиве целых чисел

  i_Max = i1[0];
  for (int cnt = 1; cnt < si1; cnt++)
  {
    if (i_Max >= i1[cnt]) continue;
    else i_Max = i1[cnt];
  }
  UpdateData(FALSE);
};
```

Программный код этого фрагмента в особых пояснениях не нуждается. Посмотрим теперь на дизассемблированный листинг C++-варианта програм-

мы, точнее на ту его часть, которая соответствует циклу `for`, вычисляющему `i_Max` (листинг 6.44).

Листинг 6.44. Программный код, полученный при дизассемблировании цикла `for`

```
       i_Max = i1[0];

0041380C  mov          eax,dword ptr [this]
0041380F  mov          ecx,dword ptr [i1]
00413812  mov          dword ptr [eax+7Ch],ecx

       for (int cnt = 1;cnt < si1; cnt++)

00413815  mov          dword ptr [cnt],1
0041381C  jmp          CFindMaxIntwithCDlg::OnBnClickedButton1+167h
(413827h)
0041381E  mov          eax,dword ptr [cnt]
00413821  add          eax,1
00413824  mov          dword ptr [cnt],eax
00413827  mov          eax,dword ptr [cnt]
0041382A  cmp          eax,dword ptr [si1]
0041382D  jge          CFindMaxIntwithCDlg::OnBnClickedButton1+18Fh
(41384Fh)

       {
         if (i_Max >= i1[cnt]) continue;

0041382F  mov          eax,dword ptr [this]
00413832  mov          ecx,dword ptr [cnt]
00413835  mov          edx,dword ptr [eax+7Ch]
00413838  cmp          edx,dword ptr i1[ecx*4]
0041383C  jl           CFindMaxIntwithCDlg::OnBnClickedButton1+180h
(413840h)
0041383E  jmp          CFindMaxIntwithCDlg::OnBnClickedButton1+15Eh
(41381Eh)

         else i_Max = i1[cnt];

00413840  mov          eax,dword ptr [this]
00413843  mov          ecx,dword ptr [cnt]
```

```
00413846  mov         edx,dword ptr i1[ecx*4]
0041384A  mov         dword ptr [eax+7Ch],edx


       }


0041384D  jmp         CFindMaxIntwithCDlg::OnBnClickedButton1+15Eh
(41381Eh)
```

Сравните дизассемблированный фрагмент кода, вычисляющего максимум, и функцию `findMax` в ассемблерном варианте программы. Даже беглого взгляда достаточно, чтобы понять избыточность второго варианта. Прежде чем проанализировать дизассемблированный код, рассмотрим еще один пример — сортировку массива целых чисел по убыванию.

Это приложение также использует диалоговое окно и два поля редактирования. В одно поле редактирования будет выводиться исходный (неупорядоченный) массив целых чисел, в другом поле можно будет видеть тот же массив, но упорядоченный по убыванию. Сортировку массива выполняет функция `sortMax`, принимающая в качестве параметров адрес массива и его размер. Программные элементы обработчика нажатия кнопки во многом схожи с программным кодом предыдущих примеров, и останавливаться на них мы не будем. Программный код функции `sortMax` и обработчика кнопки, в котором вызывается эта функция, приведен в листинге 6.45.

Листинг 6.45. Фрагменты программы, в которых выполняется сортировка и вывод результата на экран

```
void CSortarraybyMaximumDlg::sortMax(int* pi1, int si1)
{
  int isize = si1;
  _asm {
        push    EBX
        mov     EDI, DWORD PTR pi1
        mov     EBX, EDI
      big_loop:
        mov     ECX, DWORD PTR isize
        mov     EAX, [EDI]
      next:
        mov     EAX, [EDI]
        cmp     EAX, [EDI+4]
        jl      change
```

```
        jmp     cont
      change:
        xchg    EAX, [EDI+4]
        mov     [EDI], EAX
      cont:
        add     EDI, 4
        loop    next
        dec     isize
        cmp     isize, 0
        je      ex
        mov     EDI, EBX
        jmp     big_loop
      ex:
        pop     EBX
  }
}

void CSortarraybyMaximumDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  CString s1;
  int i1[] = {4, -6, -9, -7, 34, -6, 97, -50};
  int si1 = sizeof(i1)/4;
  s1.Empty;
  for (int cnt = 0; cnt < si1; cnt++)
  {
    s1.Format("%d", i1[cnt]);
    s_Src = s_Src + " " + s1;
  };
  sortMax(i1, si1);
  s1.Empty;
  for (int cnt = 0; cnt < si1; cnt++)
  {
    s1.Format("%d", i1[cnt]);
    s_Dst = s_Dst + " " + s1;
   };
UpdateData(FALSE);
}
```

Окно работающего приложения изображено на рис. 6.19.

Рис. 6.19. Окно приложения, выполняющего сортировку массива целых чисел по убыванию

Решим ту же задачу сортировки массива при помощи программы, написанной на "чистом" C++, и рассмотрим дизассемблированный листинг, точнее, тот его фрагмент, который выполняет сортировку. Фрагменты программного кода на C++, с помощью которых выполняется сортировка и вывод результата, представлены в листинге 6.46. Жирным текстом выделен интересующий нас участок программного кода.

Листинг 6.46. Программный код для выполнения сортировки массива целых чисел с использованием только операторов C++

```
void CSortbyDecreasewithCNETDlg::OnBnClickedButton1()
{

  // TODO: Add your control notification handler code here

  CString s1;
  int i1[] = {4, -6, -9, -7, 34, -6, 97, -50};
  int itmp;
  int size_i1 = sizeof(i1)/4;

  s_Src.Empty;
  s_Dst.Empty;
  UpdateData(FALSE);

  s1.Empty;

  // Вывод элементов исходного массива в поле редактирования

  for (int cnt = 0; cnt < size_i1; cnt++)
```

```
  {
    s1.Format("%d", i1[cnt]);
    s_Src = s_Src + " " + s1;
  };

  // Сортировка массива

  int tSize_i1 = size_i1;
  while (tSize_i1 != 0)
  {
    for (int cnt = 0; cnt < tSize_i1; cnt++)
    {
      if (i1[cnt] >= i1[cnt+1]) continue;
      else
      {
        itmp = i1[cnt];
        i1[cnt] = i1[cnt+1];
        i1[cnt+1] = itmp;
      };
    };
    tSize_i1--;
  };

  // Вывод элементов отсортированного массива в поле редактирования

  for (int cnt = 0; cnt < size_i1;cnt++)
  {
    s1.Format("%d", i1[cnt]);
    s_Dst = s_Dst + " " + s1;
  };
  UpdateData(FALSE);
}
```

Не будем детально останавливаться на анализе выделенного фрагмента кода, поскольку для программистов на C++ он достаточно очевиден. Дизассемблированный фрагмент этого участка кода представлен в листинге 6.47.

Листинг 6.47. Программный код дизассемблированного фрагмента на C++

```
      int tSize_i1 = size_i1;

0041382D  mov          eax,dword ptr [size_i1]
00413830  mov          dword ptr [tSize_i1],eax

      while (tSize_i1 != 0)

00413833  cmp          dword ptr [tSize_i1],0
00413837  je           CSortbyDecreasewithCNETDlg::OnBnClickedButton1+1E2h
(4138B2h)

      {
        for (int cnt = 0; cnt < tSize_i1; cnt++)

00413839  mov          dword ptr [cnt],0
00413843  jmp          CSortbyDecreasewithCNETDlg::OnBnClickedButton1+184h
(413854h)
00413845  mov          eax,dword ptr [cnt]
0041384B  add          eax,1
0041384E  mov          dword ptr [cnt],eax
00413854  mov          eax,dword ptr [cnt]
0041385A  cmp          eax,dword ptr [tSize_i1]
0041385D  jge          CSortbyDecreasewithCNETDlg::OnBnClickedButton1+1D7h
(4138A7h)
        {
          if (i1[cnt] >= i1[cnt+1]) continue;

0041385F  mov          eax,dword ptr [cnt]
00413865  mov          ecx,dword ptr [cnt]
0041386B  mov          edx,dword ptr i1[eax*4]
0041386F  cmp          edx,dword ptr [ebp+ecx*4-44h]
00413873  jl           CSortbyDecreasewithCNETDlg::OnBnClickedButton1+1A7h
(413877h)
00413875  jmp          CSortbyDecreasewithCNETDlg::OnBnClickedButton1+175h
(413845h)
```

```
          else
          {
            itmp = i1[cnt];

00413877  mov         eax,dword ptr [cnt]
0041387D  mov         ecx,dword ptr i1[eax*4]
00413881  mov         dword ptr [itmp],ecx

            i1[cnt] = i1[cnt+1];

00413884  mov         eax,dword ptr [cnt]
0041388A  mov         ecx,dword ptr [cnt]
00413890  mov         edx,dword ptr [ebp+ecx*4-44h]
00413894  mov         dword ptr i1[eax*4],edx

            i1[cnt+1] = itmp;
00413898  mov         eax,dword ptr [cnt]
0041389E  mov         ecx,dword ptr [itmp]
004138A1  mov         dword ptr [ebp+eax*4-44h],ecx

          };
        };

004138A5  jmp         CSortbyDecreasewithCNETDlg::OnBnClickedButton1+175h
(413845h)

        tSize_i1--;

004138A7  mov         eax,dword ptr [tSize_i1]
004138AA  sub         eax,1
004138AD  mov         dword ptr [tSize_i1],eax

      };

004138B0  jmp         CSortbyDecreasewithCNETDlg::OnBnClickedButton1+163h
(413833h)
```

У этих двух дизассемблированных фрагментов кода есть общие черты. Если вы заметили, C++-варианты программ для работы с переменными активно

используют основную память и значительно реже — регистры процессора. Ничего плохого в этом нет, однако это ведет к избыточности программного кода и замедлению быстродействия программы в целом. Это незаметно при небольших объемах вычислений и сравнительно небольших объемах данных, подверженных обработке. Однако при больших размерах тех же массивов данных снижение производительности станет заметным.

Например, обмен значений двух элементов массива в реализации на языке C++:

```
itmp = i1[cnt];
i1[cnt] = i1[cnt+1];
i1[cnt+1] = itmp;
```

представлен следующим эквивалентом ассемблерного кода:

```
mov         eax,dword ptr [cnt]
mov         ecx,dword ptr i1[eax*4]
mov         dword ptr [itmp],ecx
mov         eax,dword ptr [cnt]
mov         ecx,dword ptr [cnt]
mov         edx,dword ptr [ebp+ecx*4-44h]
mov         dword ptr i1[eax*4],edx
mov         eax,dword ptr [cnt]
mov         ecx,dword ptr [itmp]
mov         dword ptr [ebp+eax*4-44h],ecx
```

В то же время, используя комбинацию команд ассемблера:

```
mov     EAX, [EDI]
xchg    EAX, [EDI+4]
mov     [EDI], EAX
```

можно добиться повышения производительности в программе поиска максимума. В этом случае значения переменных хранятся в регистрах, и вычисления выполняются очень быстро, т. к. значительно уменьшен обмен данными по системной шине.

То же самое относится и к оптимизации циклов. Объективно заставить компилятор C++ сгенерировать программный код с максимальным использованием регистров вместо памяти очень трудно, а во многих случаях не-

возможно. Хорошо спроектированные на ассемблере циклические вычисления выполняются существенно быстрее и требуют меньшего числа команд.

Рассмотрим, как работает цикл `for` в программе сортировки. Оператор:

```
for (int cnt = 0; cnt < tSize_i1; cnt++)
```

распадается на несколько команд ассемблера. Дизассемблированный код этого оператора модифицируем так, чтобы он воспринимался легче. Для этого уберем ссылки на адреса физической памяти в командах переходов и в соответствующих местах заменим их метками. Модифицированный код будет выглядеть так:

```
  mov     dword ptr [cnt], 0
  jmp     L2
L1:
  mov     eax,dword ptr [cnt]
  add     eax,1
  mov     dword ptr [cnt], eax
L2:
  mov     eax,dword ptr [cnt]
  cmp     eax,dword ptr [tSize_i1]
  jge     <адрес>
          \
  < операторы цикла >

  jmp     L1
   ...
```

Нельзя сказать, что этот код неоптимален. Если бы вы имели в своем распоряжении только регистры процессора EAX, EDX и ECX, то для организации цикла `for` использовали бы, скорее всего, тот же самый алгоритм. В силу этих ограничений пришлось бы хранить переменные циклов в памяти, и каждый раз (как в этом фрагменте кода) извлекать их для очередной итерации.

Если использовать ассемблер, то реализация оператора `for` при помощи стандартного алгоритма с использованием регистра ECX:

```
mov     ECX, dword ptr isize
  ...
loop    next
```

выполняется быстрее.

Как видите, применение языка ассемблер способно решить многие проблемы оптимизации программы, но только если вы хорошо представляете себе, что оптимизировать и как. Это касается не только C++ .NET, Delphi 7, но и других компиляторов.

Еще один важный момент. Современные компиляторы очень мало используют возможности новых архитектур процессоров и их систем команд. Это очень обширная тема, которая требует отдельного обсуждения, но здесь кроются большие резервы для программиста.

Встроенный ассемблер можно с успехом применить и при обработке строк. Несмотря на то, что среда разработки C++ .NET имеет мощные процедуры обработки строк, использование ассемблера оказывается эффективным и здесь. Дело в том, что часто требуется специфичная обработка строковых переменных, и реализация такой обработки стандартными процедурами оказывается весьма громоздкой и медленной. Вначале рассмотрим наиболее широко используемые типы строк и методы их конвертации.

Как и во всех языках высокого уровня, в C++ .NET широко используются строки с завершающим нулем. Для манипуляции с такими строками в этой среде программирования разработано много самых разнообразных функций. Оптимизация обработки таких строк при помощи ассемблерных процедур была рассмотрена нами в *главе 3*.

Однако в C++ .NET используются и другие типы строк. Сложность манипуляций со строками с завершающим нулем привела разработчиков Microsoft к необходимости создания класса `CString`. Этот класс стал весьма популярным среди программистов. Строка типа `CString` представляет собой последовательность символов переменной длины. Символы строки могут быть как 16-битовые (кодировка `UNICODE`), так и 8-битовые (кодировка `ANSI`). Для манипуляции со строками используются методы и свойства класса `CString`. Этот класс имеет мощные функции для работы со строками, превосходящие по своим возможностям некоторые стандартные функции языка C++, такие как `strcat` или `strcopy`.

Для инициализации `CString` объекта можно использовать оператор `CString`:

```
CString s = "Это строка типа CString";
```

Можно присвоить значение одного объекта `CString` другому:

```
CString s1 = "Это тестовая строка";
CString s2 = s1;
```

При такой операции содержимое s1 копируется в s2. Для конкатенации двух и более строк можно использовать операторы "+" или "+=":

```
CString s1 = "Строка 1";
CString s2 = "Строкой 2";
s1 += " объединяется ";
CString sres= s1 + "со " + s2;
```

В результате выполнения этой последовательности получим результирующую строку:

```
Строка 1 объединяется со Строкой 2
```

Чтобы манипулировать с отдельными элементами строки CString, можно использовать функции GetAt и SetAt этого класса. Первый элемент строки всегда имеет индекс 0. Например, чтобы получить символ строки s1 с индексом 3, имеющей значение "СТРОКА 1", можно выполнить следующий оператор:

```
s1.GetAt(3)
```

Такого же результата можно добиться, используя оператор "[]". Тогда доступ к элементу строки будет выглядеть так же, как и к элементу массива:

```
s1[3]
```

Результатом операции будет символ "О". Чтобы поместить в позицию с индексом 5 (6-й элемент) этой же строки символ "И", необходимо выполнить оператор:

```
s1.SetAt(5, 'И')
```

Самой мощной функцией класса CString является функция Format. Она позволяет преобразовать данные других типов в текст и напоминает стандартные функции sprintf и wsprintf. В предыдущих примерах мы применяли эту функцию для вывода элементов массива в поле редактирования Edit. Приведу небольшой фрагмент программного кода:

```
for (int cnt = 0; cnt < size_i1; cnt++)
```

```
{
  s1.Format("%d", i1[cnt]);
  s_Src = s_Src + " " + s1;
};
```

Этот код используется для вывода элементов целочисленного массива i1 в поле редактирования Edit. Элемент управления Edit имеет тип CString, т. к. связан с переменной s_Src типа CString. Здесь же присутствует вспомогательная переменная s1, имеющая такой же тип, которую мы используем для преобразования целочисленного элемента массива в строковый тип. Оператор:

```
s_Src = s_Src + " " + s1;
```

нам знаком и применяется для вывода преобразованных элементов массива на экран.

Как видите, класс CString во многом упрощает работу со строками (мы рассмотрели только малую часть его возможностей!). Каким же образом можно манипулировать объектами CString, используя встроенный ассемблер?

Лучше всего продемонстрировать это на примере. Рассмотрим следующую задачу: требуется в строке типа CString заменить все символы пробела символами "+" и вывести результат преобразования на экран.

Для решения задачи разработаем приложение на основе диалогового окна и разместим на нем три элемента Edit, кнопку Button и три метки статического текста Label. Поставим в соответствие элементу Edit1 переменную s1 типа CString, элементу Edit2 — переменную s2 типа CString и, наконец, элементу Edit3 — переменную length_s1 целого типа.

В поле редактирования Edit1 будет вводиться исходная строка с пробелами, в поле Edit2 будет выведен результат замены пробелов на символы "+", а в поле Edit3 будет отображен размер строки.

Вначале рассмотрим фрагмент программного кода для обработки строки, написанный на C++ .NET (листинг 6.48).

Листинг 6.48. Обработчик нажатия кнопки, в котором выполняется обработка строки CString с помощью операторов C++ .NET

```
void CReplacecharinStringDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here
```

```
  UpdateData(TRUE);
  LPSTR lps2;

  s_Len = strlen((LPCTSTR)s1);
  s2 = s1;
  lps2 = s2.GetBuffer(128);
  for (int cnt = 0; cnt < s_Len; cnt++)
  {
    if (*lps2 == ' ') *lps2 = '+';
    lps2++;
  }
  UpdateData(FALSE);
  s2.ReleaseBuffer;
}
```

Для доступа к произвольному элементу строки или массива, как известно, необходимо знать адрес этого массива, его размер и тип элементов, входящих в этот массив. Для строк с завершающим нулем адресом строки является адрес первого элемента. Доступ к элементам строки выполняется через индексирование адреса строки.

Чтобы получить доступ к элементам строки `CString`, можно воспользоваться функцией `GetBuffer`, передав ей в качестве параметра размер буфера памяти. В данном случае 128 байт вполне достаточно. Результатом выполнения этой функции является указатель на буфер в памяти, что позволяет работать с отдельными элементами так же, как и в обычных функциях обработки строк. Воспользовавшись парой операторов:

```
  ...
LPSTR lps2;
  ...
lps2 = s2.GetBuffer(128);
  ...
```

получим адрес буфера строки. Остается определить размер строки. Особых проблем здесь тоже не возникает, достаточно воспользоваться классической функцией `strlren` и сохранить результат в переменной `s_Len`:

```
s_Len = strlen((LPCTSTR)s1);
```

Далее остается выполнить поиск символов пробела в буфере строки и заменить их символом "+". Это делается при помощи цикла `for`. После выполнения всех манипуляций необходимо освободить буфер:

```
s2.ReleaseBuffer;
```

Окно работающего приложения изображено на рис. 6.20.

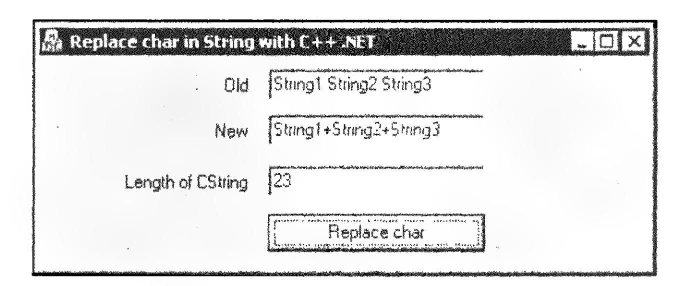


Рис. 6. 20. Окно приложения, выполняющего замену символа пробела на символ плюс в строке типа `CString`

Можно оптимизировать предыдущую программу, заменив цикл `for` компактной ассемблерной процедурой. Исходный текст процедуры на ассемблере (назовем ее `replaceChar`) приведен в листинге 6.49.

Листинг 6.49. Функция на ассемблере, выполняющая поиск и замену символов в строке типа `CString`

```
void CReplaceCharinCStringwithBASMDlg::replaceChar(char* ps1, int ls1)
{
  _asm {
        mov     EDI, ps1
        mov     ECX, ls1
        cld
        mov     AL, ' '
      next:
        scasb
        je      change
      cont:
        loop    next
        jmp     ex
      change:
        mov     [EDI-1], '+'
```

```
        jmp     cont
      ex:
   };
}
```

Процедура принимает в качестве параметров адрес буфера строки и размер строки. Адрес буфера загружается в регистр EDI, а размер строки в регистр ECX . Для поиска и замены символов воспользуемся строковой командой scasb, которая сравнивает содержимое регистра AL (символ пробела) с текущим элементом строки. Количество итераций определяется размером строки. Так как после сравнения значение адреса было увеличено на 1, то если пробел найден, он заменяется на символ "+" с помощью команды:

```
mov     [EDI-1], '+'
```

Исходный текст обработчика нажатия кнопки с учетом сделанных изменений приведен в листинге 6.50.

Листинг 6.50. Использование ассемблерной процедуры для поиска и замены символов в строке типа CString

```
void CReplaceCharinCStringwithBASMDlg::OnBnClickedButton1()
{
  // TODO: Add your control notification handler code here

  UpdateData(TRUE);
  LPSTR lps2;
  length_s1 = strlen((LPCTSTR)s1);
  s2 = s1;
  lps2 = s2.GetBuffer(128);
  replaceChar(lps2, length_s1);
  UpdateData(FALSE);
  s2.ReleaseBuffer;
};
```

На этом анализ возможностей встроенного ассемблера двух наиболее популярных средств разработки Delphi 7 и Visual C++ .NET можно закончить. По возможности попытаемся охватить ключевые моменты применения ассемблера в приложениях. Главное внимание было уделено технике применения встроенного ассемблера на практике при работе с различными типами данных.

Заключение

Автор попытался дать в книге обширную информацию по применению ассемблера для программирования Windows-приложений. В конце хотелось бы сделать некоторые дополнительные замечания, касающиеся материала книги. Автор постарался охватить как можно более широкий диапазон вопросов по данной теме, хотя некоторые из них остались незатронутыми. Все же можно надеятся, что даже в таком объеме книга принесет пользу читателю.

Так сложилось, что отечественные программисты, пишущие на ассемблере, большую часть своих программ разрабатывают на одном из двух языков: либо макроассемблере MASM фирмы Microsoft, либо на Турбо ассемблере фирмы Borland. Большая часть примеров книги разработана с использованием MASM. Хотелось бы рекомендовать программистам обратить внимание и на другие инструменты разработки на языке ассемблера, например NASM.

На ассемблере можно создавать программы любой сложности, если уметь использовать возможности, предоставляемые операционной системой. Windows, как вы убедились, обладает мощным интерфейсом, созданным специально для программиста и включающим в себя сотни функций WIN API. Успех в создании вашей программы (это касается не только языка ассемблера) во многом зависит от умения использовать этот интерфейс. Из многочисленных примеров видно, что программировать на ассемблере в Windows одновременно и легче, и труднее по сравнению с MS-DOS. Легче потому, что вам не нужно заботиться о правильной инициализации сегментных регистров и выборе моделей памяти. Труднее потому, что архитектура Windows требует от программиста иных подходов, чем традиционная MS-DOS.

Современные средства разработки на ассемблере, такие как MASM 32 или AsmStudio, позволяют создавать приложения на языке ассемблера довольно быстро и качественно. При всех достоинствах этих пакетов разработки следует отметить и один существенный недостаток: ни фирма Microsoft, ни фирма Borland больше не развивают автономные компиляторы

ассемблера и прекратили разработки в этом направлении. Альтернативой, причем довольно неплохой, являются продукты сторонних фирм и независимых разработчиков, например тот же NASM. Одной из причин отказа крупных фирм от разработок компиляторов ассемблера является то, что ассемблер стал частью среды программирования в языках высокого уровня.

Встроенный ассемблер языков высокого уровня хоть и не является самостоятельным средством разработки, но весьма эффективен для написания быстрых и производительных программ. Хотелось бы надеяться, что приведенные примеры для Delphi 7 и Visual C++ .NET смогли убедить читателей в необходимости использования ассемблера в своих программах.

Хотелось бы также верить, что эта книга станет настольной для многих программистов — как опытных, так и начинающих.

Приложение 1

Инструкции процессоров 80×86

Это приложение является справочником по системе команд семейства процессоров Intel. В справочник включены команды для моделей процессоров 80386 и более поздних. Для описания форматов команд используется ряд аббревиатур, представленных в табл. П1.1. Сами команды описаны в табл. П1.2.

Таблица П1.1. *Аббревиатуры для описания команд*

Обозначение	Краткое описание
reg	Один из 8-, 16- или 32-разрядных регистров из списка: AH, AL, BH, BL, CH, CL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
reg8, reg16, reg32	Регистр общего назначения, определяемый количеством битов
acc	AL, AX или EAX
mem	Операнд в памяти
mem8, mem16, mem32	Операнд в памяти, определяемый количеством битов
immed	Непосредственный операнд
immed8, immed16, immed32	Непосредственный операнд с определенным количеством битов
label	Метка

Таблица П1.2. Система команд

Код операции	Операнды	Функция
aaa		ASCII коррекция после сложения
aad		ASCII коррекция перед делением
aam		ASCII коррекция после умножения
aas		ASCII коррекция после вычитания
adc	reg, reg mem, reg reg, mem reg, immed mem, immed acc, immed	Сложение с переносом
add	reg, reg mem, reg reg, mem reg, immed mem, immed acc, immed	Сложение
and	reg, reg mem, reg reg, mem reg, immed mem, immed acc, immed	Логическое "И"
bsf, bsr	reg16, reg16 reg16, mem16 reg32, reg32 reg32, mem32	Сканирование битов
bt, btc, btr, bts	reg16, immed8 reg16, reg16 mem16, immed8 mem16, reg16	Проверка битов

Таблица П1.2 *(продолжение)*

Код операции	Операнды	Функция
call	label reg mem16 mem32	Вызов процедуры
cbw		Преобразование байта в слово
cdq		Преобразование двойного слова в учетверенное
clc		Сбросить флаг переноса
cld		Сбросить флаг направления
cli		Сбросить флаг прерывания
cmc		Инвертировать флаг переноса
cmp		Сравнение операндов
cmps, cmpsb, cmpsw, cmpsd	mem, mem	Сравнение строк
cwd		Преобразовать слово в двойное слово
daa		Десятичная коррекция после сложения
das		Десятичная коррекция после вычитания
dec	reg mem	Декремент
div	reg mem	Деление без знака
idiv	reg mem	Деление целых чисел со знаком
imul	reg mem	Умножение целых чисел со знаком
in	acc, immed	Ввод из порта
inc	reg mem	Инкремент
int		Генерирует программное прерывание
iret		Возврат из прерывания

Таблица П1.2 *(продолжение)*

Код операции	Операнды	Функция
Jcondition	label	Переход, если выполнено условие
jmp	label	Безусловный переход
lahf		Загрузка флагов в AH
lds, les, lfs, lgs, lss		Загрузка дальнего указателя
lea	reg, mem	Загрузка текущего адреса
lods, lodsb, lodsw, lodsd	mem	Загрузка строки в аккумулятор
loop	label	Цикл, декрементирует регистр CX и переход на метку, пока CX больше 0
loope, loopz	label	Цикл, если равно 0. Декрементирует регистр CX и выполняет переход на метку, если CX больше 0 и флаг нуля установлен
loopne, loopz	label	Цикл, если не равно 0. Декрементирует регистр CX и выполняет переход на метку, если CX больше 0 и флаг нуля сброшен
mov	reg, reg mem, reg reg, mem reg, immed mem, immed	Пересылка операндов
movs, movsb, movsw, movsd	mem, mem	Пересылка строк
mul	reg mem	Умножение целых чисел без знака
neg	reg mem	Изменить знак операнда
nop		Не производит никаких операций, используется для задержек во временных циклах
not	reg mem	Логическая функция "НЕ", инвертирует каждый бит операнда

Таблица П1.2 (продолжение)

Код операции	Операнды	Функция
or	reg, reg mem, reg reg, mem reg, immed mem, immed acc, immed	Логическое "ИЛИ"
out	immed, acc DX, acc	Вывод в порт
pop	reg16 reg32 mem16 mem32	Извлечение операнда из стека
popa, popad		Извлечение из стека регистров общего назначения (popa — 16-разрядных, popad — 32-разрядных)
popf, popfd		Извлечение флагов из стека
push	reg16 reg32 mem16 mem32	Поместить в стек
pusha, pushad		Помещает в стек все регистры
pushf, pushfd		Помещение регистра флагов в стек
rcl	reg, immed8 reg, CL mem, immed8 mem, CL	Циклический сдвиг операнда влево через флаг переноса

Таблица П1.2 (продолжение)

Код операции	Операнды	Функция
rcr	reg, immed8 reg, CL mem, immed8 mem, CL	Циклический сдвиг операнда вправо через флаг переноса
rep		Повторить команду для строкового примитива, используя регистр CX как счетчик
repcondition		Повторять команды строковых примитивов по условию
ret		Возврат из процедуры
retn	immed8	Возврат из процедуры с восстановлением стека. Непосредственный операнд определяет значение, которое должно быть добавлено к регистру-указателю стека
rol	reg, immed8 reg, CL mem, immed8 mem, CL	Циклический сдвиг влево
ror	reg, immed8 reg, CL mem, immed8 mem, CL	Циклический сдвиг вправо
sahf		Загрузка регистра флагов из регистра AH
sal	reg, immed8 reg, CL mem, immed8 mem, CL	Арифметический сдвиг влево

Таблица П1.2 (продолжение)

Код операции	Операнды	Функция
sar	reg, immed8 reg, CL mem, immed8 mem, CL	Арифметический сдвиг вправо
sbb	reg, reg mem, reg reg, mem reg, immed mem, immed	Вычитание с заемом
scas, scasb, scasw, scasd	mem	Сканирует строку, сравнивая значения элементов со значением в аккумуляторе
SETcondition	reg8 mem8	Установка по условию. Если заданное условие истинно, то байт-получатель устанавливается в 1, если ложно — в 0
shl	reg, immed8 reg, CL mem, immed8 mem, CL	Логический сдвиг влево
Shr	reg, immed8 reg, CL mem, immed8 mem, CL	Логический сдвиг вправо
stc		Устанавливает флаг переноса
std		Устанавливает флаг направления
sti		Устанавливает флаг прерывания
stos, stosb, stosw, stosd	mem	Сохранение содержимого аккумулятора в ячейке памяти, принадлежащей буферу строки

Таблица П1.2 (продолжение)

Код операции	Операнды	Функция
sub	reg, reg mem, reg reg, mem reg, immed mem, immed acc, immed	Вычитание
test	reg, reg mem, reg reg, mem reg, immed mem, immed acc, immed	Проверка отдельных битов операнда-получателя с соответствующими битами операнда-приемника. Выполняется операция логическое "И", в результате устанавливаются соответствующие флаги
wait		Приостанавливает работу процессора
xchg	reg, reg mem, reg reg, mem	Обмен содержимого операнда-отправителя и операнда-получателя
xlat, xlatb	Mem	Использует значение в регистре AL как индекс таблицы, на которую указывает содержимое регистра BX
xor	reg, reg mem, reg reg, mem reg, immed mem, immed acc, immed	Логическое исключающее "ИЛИ"

Приложение 2

Описание CD

На прилагающемся CD записаны исходные тексты программ. Примеры размещены в каталогах Chapter_n, где n — номер главы. Помимо исходных текстов, каталоги, относящиеся к *главе 3* и *6*, содержат файлы проектов для MS Visual C++ .NET и Delphi 7. В каждом таком каталоге имеется текстовый файл Readme.doc, в котором приводится описание содержимого каталога.

Для компиляции исходных текстов программ необходимо иметь установленное на персональном компьютере соответствующее программное обеспечение (MASM, TASM 5.0, Microsoft Visual C++ .NET, Delphi 7). Желательно также установить последние пакеты обновления для Visual Studio .NET и Delphi 7.

Список литературы

1. Жуков А. В., Авдюхин А. А. Ассемблер. — СПб.: БХВ-Петербург, 2002.
2. Ирвин К. Язык ассемблера для процессоров Intel. 3-е изд. / Пер. с англ. — М.: Издательский дом "Вильямс", 2002.
3. Использование Turbo Assembler при разработке программ: Учебное издание. — Киев: Диалектика, 1994.
4. Оберг Р. Дж., Торстейнсон П. Архитектура .NET и программирование с помощью Visual C++ / Пер. с англ. — М.: Издательский дом "Вильямс", 2002.
5. Саймон Р. Windows 2000 API. Энциклопедия программиста / Пер. с англ. — К.—СПб.: ООО "ДиаСофтЮП", 2002.
6. Сван Т. Освоение Turbo Assembler. — Киев: Диалектика, 1996.
7. Финогенов К. Г. Самоучитель по системным функциям MS-DOS. 2-е изд., перераб. и дополн. — М.: Радио и связь, Энтроп, 1995.
8. Шилдт Г. Программирование на С и С++ для Windows 95. — Киев: Торгово-издательское бюро BHV, 1996.

Предметный указатель

C

M

O

А

Б

В

Д

И

К

Л

М

О

П

Р

С

У

Ф

Ц

Ч



Магда Юрий Степанович, специалист по системам обработки данных, имеет диплом системного инженера UNIX, автор публикаций в журналах "Радиоаматор" (Украина), "Circuit Cellar" (США), "Electronic Design" (США).



Технология создания эффективного кода

Ассемблер

Разработка и оптимизация Windows-приложений

Подробное руководство знакомит читателей с различными вариантами оптимизации программ. Автор рассматривает применение ассемблера и как самостоятельного средства разработки полнофункциональных Windows-приложений, и как встроенного в составе языков высокого уровня. Ряд возможностей ассемблера описываются впервые. Материал книги включает много примеров с анализом программного кода. Все примеры программ работоспособны и построены таким образом, чтобы их можно было легко адаптировать или модифицировать для дальнейшего использования. Книга будет полезна и программистам, работающим с языками высокого уровня, и программистам, пишущим на ассемблере.



Компакт-диск содержит примеры приложений и тексты программ


ISBN 5-94157-324-3
9 785941 573240

Уровень пользователя	Средний/высокий
Категория	Программирование

БХВ-Петербург 198005, Санкт-Петербург, Измайловский пр., 29
E-mail: mail@bhv.ru Internet: www.bhv.ru Тел.: (812) 251-4244 Факс: (812) 251-1295